Project Number: GFP 1203

Software Process Configurator, v2

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Wesley Ripley

Adam Talbot

Date: March 1, 2013

Approved:

Keywords:

1. Software Engineering
2. Software Process
3. Decision Support Systems

Professor Gary F. Pollice, Major Advisor

Abstract

The Software Process Configurator is a tool that suggests development methodologies for new software projects based on the given project's characteristics. These characteristics are accumulated through a dialogue-based interface. The decision process operates from a collection of expert recommendations about software development that can be modified by software development experts using a domain specific language.

Acknowledgments

We would like to thank Professor Gary Pollice, our advisor, for his guidance and expertise in the field of Software Engineering.

We would also like to thank Pragathi Balasubramanian and Jake Todaro for their prior work on this project and for providing us with the groundwork to build off of.

Table of Contents

Abstract i

Acknowledgments ii

Table of Contents iii

List of Figures v

1.0 Introduction 1

2.0 Background 3

2.1 Software Development Processes 3

2.1.1 Waterfall 3

2.1.2 Iterative and Incremental Development 4

2.1.3 Agile 6

2.1.4 Lean Thinking, Feature Driven Development, and Kanban 8

2.2 Decision Support Systems and Expert Systems 8

2.3 Domain Specific Languages 9

2.4 The Software Process Configurator 10

3.0 Methodology 12

3.1 Requirements 12

3.2 Scope 12

3.3 Goals 12

3.4 Evaluating Past Progress 13

3.4.1 Problems with Old Design 14

3.5 Designing the Recommendation System 15

3.5.1 Decoupling from Dialog system 15

3.5.2 Implementing Recommendations 16

3.5.3 Attributes and Values 17

3.5.4 Assets 19

3.5.5 Presenting Recommendations 20

3.6 Designing the DSL 21

3.6.1 How to Create Recommendations 21

3.6.2 Designing the Syntax 22

3.6.3 Implementing the DSL 25

3.7 Gathering Data 26

4.0 Results 28
4.1 Use Case – Process Engineer 28
4.2 Use Case – Process Selector 34
5.0 Future Work 35
5.1 Client-Server Database 35
5.2 Dialog Tool 35
5.3 Expert Knowledge 36
5.4 Web System 36
5.5 Improved Dialog Options 37
5.6 Improved Explanation System 37
6.0 References 38
Appendix A Glossary of Terms 40
Appendix B SPC 2.0 Unified Modeling Language (UML) Diagram 41
Appendix C Software Engineering Survey 48
Appendix D Grammar for the Domain Specific Language 55
Appendix E Recommendation Examples 57

List of Figures

Figure 1: The Waterfall Development Method (Rerych, 2002) 4
Figure 2: The Spiral Development Method (Boehm, 1986) 5
Figure 3: The Evo Development Method (May & Zimmer, 1996) 5
Figure 4: The Rational Unified Process ("IBM Rational Unified," 2007) 6
Figure 5: Extreme Programming (XP) ("XP practices," 2008) 7
Figure 6: Scrum Methodology (Lacey, 2012) 7
Figure 7: FDD Life Cycle (Ambler, 2005) 8
Figure 8: UML of SPC 1.0 (Todaro & Balasubramanian, 2012) 10
Figure 9: A Gherkin Scenario (Wynne & Hellesoy, 2012) 23
Figure 10: A "Project Manager" Recommendation 25
Figure 11: The Recommendation Tool for Domain Experts 30
Figure 12: Sample Recommendation 31
Figure 13: Adding a new Attribute 31
Figure 14: Finding a Recommendation to add 32
Figure 15: The Recommendation is loaded into the system 32
Figure 16: Sample Dialog Information 33

1.0 Introduction

Computers, technology, and software can be found almost everywhere in modern society. Software development is an ever-growing field; as of May 2011, approximately 0.97% of all jobs were in software development and computer programming, up from 0.95% in 2010 and .88% in 2001. ("Occupational employment statistics," 2013) As such, there are many different approaches to developing software. Waterfall, Agile, Lean, and Scrum are commonplace words in the realm of software development. Different companies adopt different practices based on their needs, sometimes without a complete understanding of what these practices entail.

As a result, today's software is developed in many different ways. This has both positive and negative aspects; on the positive side, each approach incorporates different mechanisms that are suitable for different types of software projects. Because there are so many options, software developers have the opportunity to pick and choose to construct a strategy that works well for their purposes. The problem is that when developing a new project, there is often not enough time to find expert opinions on what methods are best suited for a particular project. In other words, project leaders cannot take advantage of these development options without spending time that could otherwise be used for development.

In order to solve this problem, the Software Process Configurator (SPC) was developed. It is targeted towards those who are planning small-scale projects, and has the potential to be expanded to solve the problems that large companies face. The general approach of the SPC is to educate these users by completing the following objectives:

1. Allow users to provide relevant information regarding their software project,
2. Generate a report that recommends software development techniques, methods, and tools depending on the characteristics of their project, and
3. Provide resources that allow users to learn how to implement the suggested recommendations.

To satisfy these objectives, the SPC was designed with the following guidelines:

1. Users should be given a guided dialog to simplify the entry of information, and the questions should correspond to characteristics of the project,
2. Experts should be able to modify the collection of recommendations that can be given to the user upon completing the dialogue, and

3. The underlying decisioning framework must be reusable for other purposes.

The end result of this project was an Alpha version of the Software Process Configurator, which we will refer to as SPC 2.0. It was developed from a proof of concept (SPC 1.0) completed in 2012. SPC 2.0 uses a dialog system to collect a set of constraints on the project, and these constraints are compared to the recommendations supplied by Process Engineers. These recommendations are added by using a domain specific language to write up a specification, which is then loaded into a tool that connects to the database. After the user, or Process Selector, completes the dialog, the system generates an HTML page that provides an ordered list of all matching recommendations. Each recommendation comes with a description, an explanation for why it was chosen, and a collection of assets the Process Selector can use to learn more. The dialog, recommendations, project attributes, and the final recommendations are all designed to be editable by Process Engineers.

2.0 Background

To provide an understanding of why the Software Process Configurator (SPC) was developed, this section describes some of the current trends and practices in software development. It also discusses Decision Support Systems and Expert Systems, and why the SPC is an example of each. Because this is the second version of the SPC, this section reviews what had been accomplished in the first version. This Background should be used to help familiarize the reader with the subject area the SPC was developed for.

2.1 Software Development Processes

Software engineering was first discussed in a Software Engineering conference sponsored by the NATO Science Committee, in October 1968. (Naur & Randell, 1968) Before this time, software development was not very well structured. The NATO Science Committee recognized the problems that the up-and-coming Software Engineers would be facing, and so they discussed issues including design strategies and techniques, structure, and feedback through simulation.

Since this time, many examples of software development processes have been adopted by different companies. Each process contains many specific development practices, and individual companies may choose to adopt only certain sets of them depending on their needs. As such, it is useful to look at what practices some of the popular processes contain in order to give examples of the type of recommendations that might be given to users of the SPC.

2.1.1 Waterfall

The “Waterfall development” method involves a specific set of phases, which may include requirements analysis, design, testing, and the product release. Development flows from one stage to the next in a pre-set order, much like how a waterfall might flow down a mountainside. The waterfall development method also includes feedback loops between each of the stages, which allow the developers to go back to the previous stage and fix any new problems that may have come up. However, it is important that that stage is considered finished before moving on to the next stage.

While the Waterfall development method is simple and straightforward, it has flaws in practice. Common criticisms include that Waterfall fails in situations where the customer may not initially know the specifics of what they want, or when the requirements change often during development. Nevertheless, it can be a useful tool for certain situations when customers are not involved, and developers can choose to use bits and pieces of this methodology depending on their project’s requirements. (Melonfire, 2006)

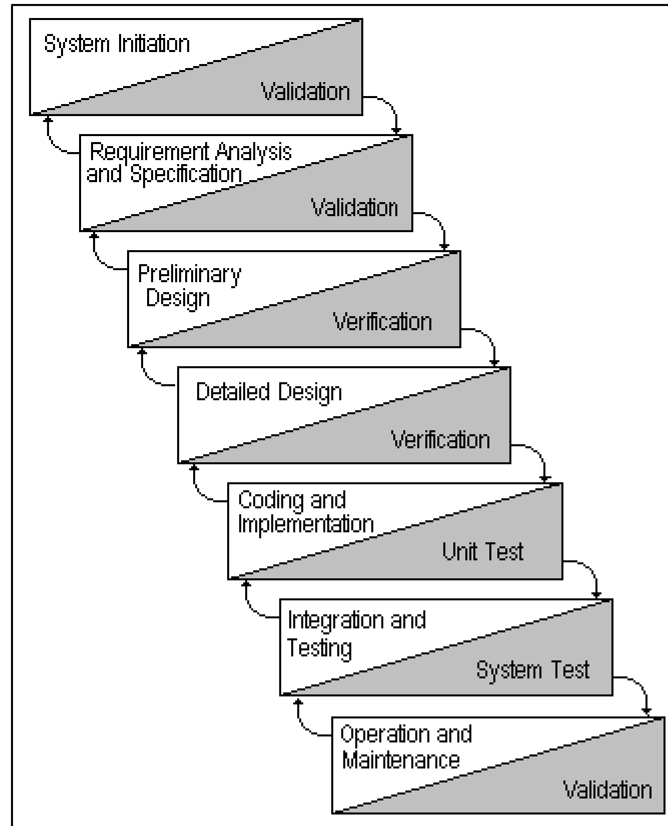


Figure 1: The Waterfall Development Method (Rerych, 2002)

2.1.2 Iterative and Incremental Development

The poor results of using Waterfall and other early methodologies led to a new generation of software development practice, which would later lead to Agile development. These practices began with the idea of iteration-based development, which accounts for changing requirements and is designed to minimize risk. Examples of these iterative processes include Spiral, Evo, and the Rational Unified Process. (Kennaley, 2010)

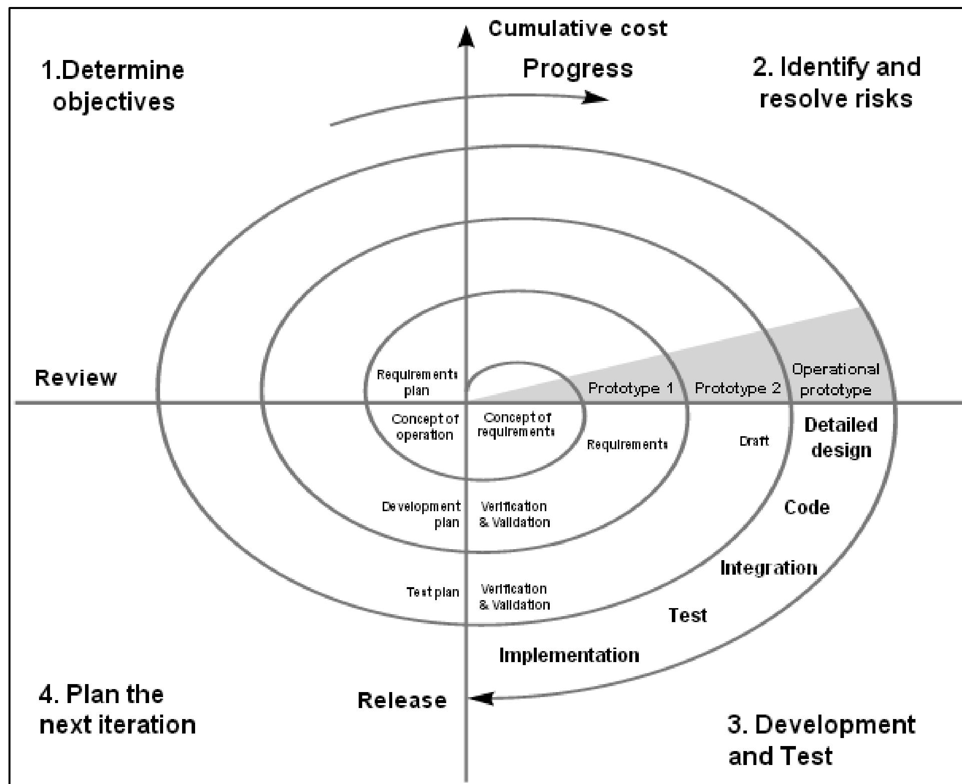


Figure 2: The Spiral Development Method (Boehm, 1986)

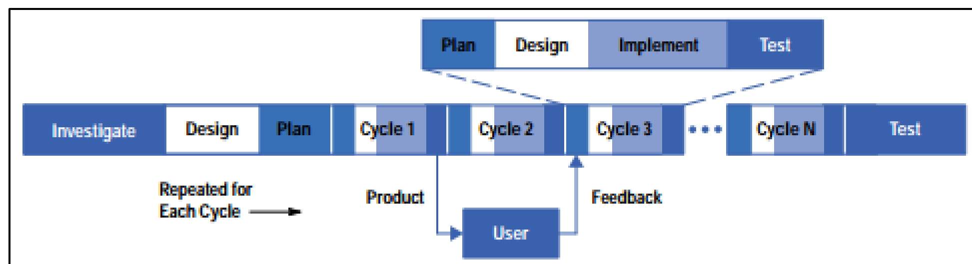


Figure 3: The Evo Development Method (May & Zimmer, 1996)

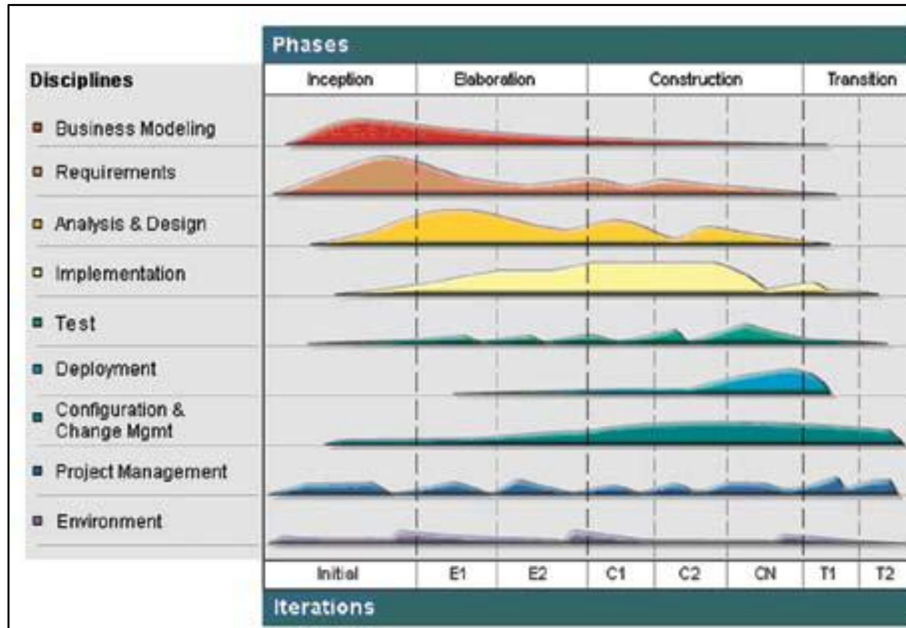


Figure 4: The Rational Unified Process ("IBM Rational Unified," 2007)

2.1.3 Agile

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over *processes and tools*

Working software over *comprehensive documentation*

Customer collaboration over *contract negotiation*

Responding to change over *following a plan*

That is, while there is value in the items on the right, we value the items on the left more." (Beck, 2001)

This is the "Agile Manifesto," a basic description of the rationale behind Agile development. Agile is more of a philosophy than a programming practice; it specifies many guidelines that the founders of Agile believed would be most practical and profitable for those who followed them. Because it is not a specific process, many companies can "be Agile" while still working in very different ways. However, there are some specific Agile-based development practices that have been adopted by various companies. These practices include Extreme Programming and Scrum.

Extreme Programming, or XP, was developed by Kent Beck after being hired to lead the Chrysler CCC project. It focuses on increased frequency of feedback and waste reduction. According to SDLC 3.0, it was the first development method considered to be "Agile." (Kennaley, 2010) Scrum did not gain

popularity until later. Unlike other methods before it, Scrum uses a "Product Backlog" to manage a queue of product requests. Like XP, there are daily stand up meetings, but they are instead run by the team itself and not the project manager, or Scrum Master. (Kennaley, 2010)

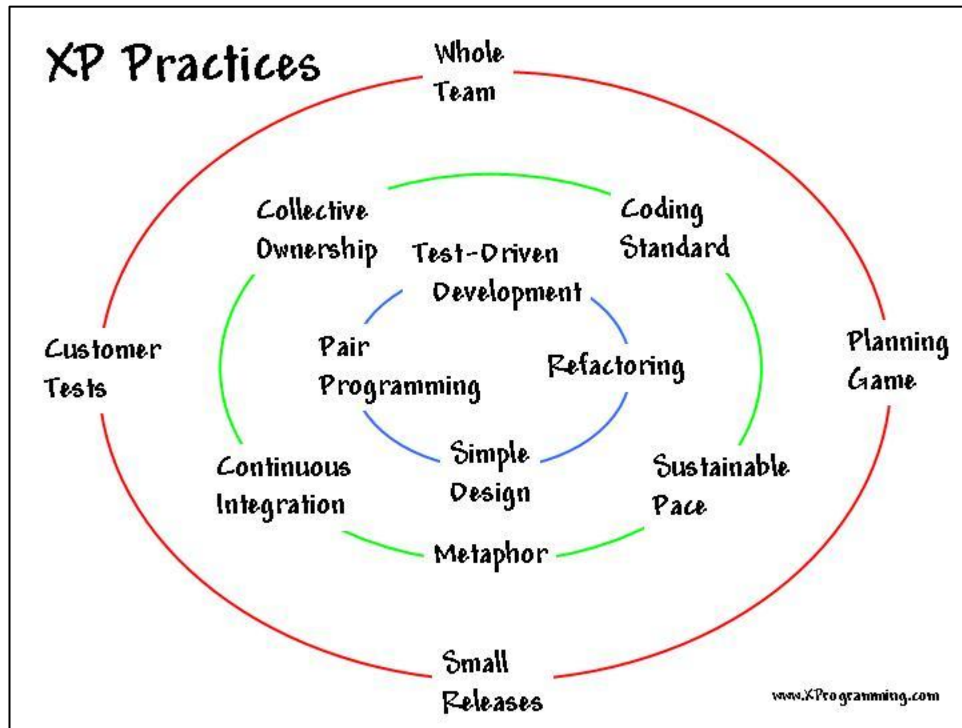


Figure 5: Extreme Programming (XP) ("XP practices," 2008)

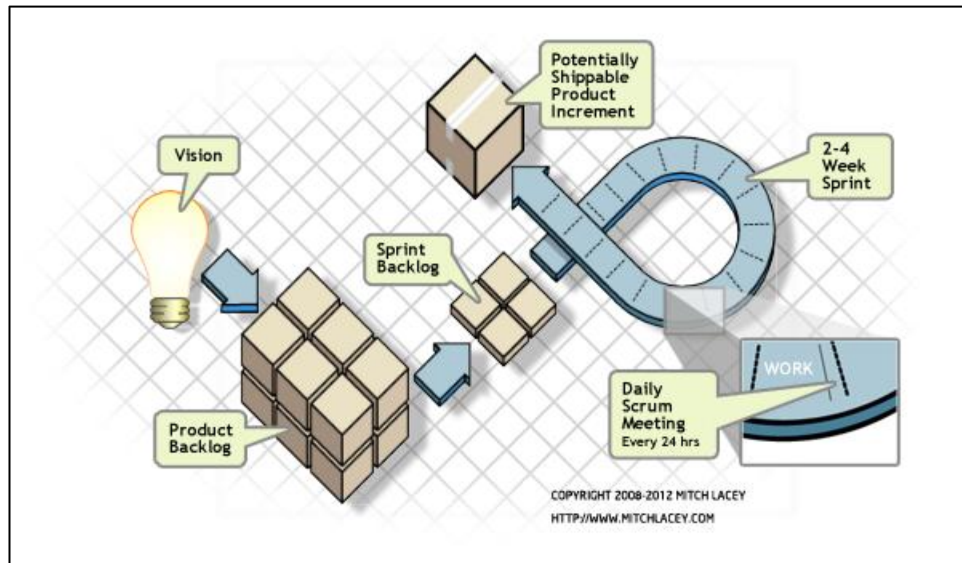


Figure 6: Scrum Methodology (Lacey, 2012)

2.1.4 Lean Thinking, Feature Driven Development, and Kanban

Not all modern software development practices are Agile. Lean Thinking developed separately from Agile, and consists of 22 management tools. These tools are sorted into seven categories: Eliminate waste, Amplify learning, Decide as late as possible, Deliver as fast as possible, Empower the team, Build integrity in, and See the whole. Feature Driven Development, or FDD, emerged before Agile and is referred to as an "iteration-less" development practice. (Kennaley, 2010) Kanban development, also similar to Lean, is a result of studies of Feature Driven Development and focuses on queuing theory. (Kennaley, 2010)

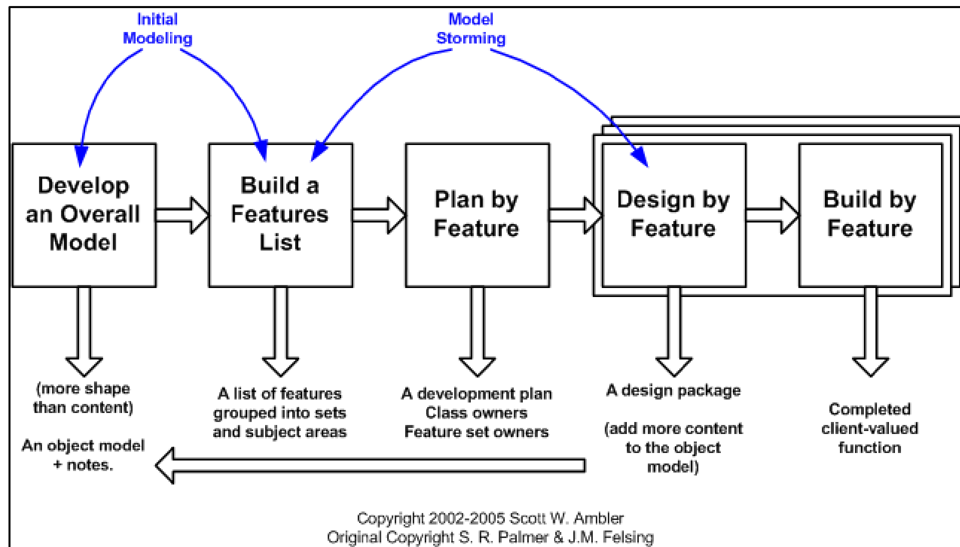


Figure 7: FDD Life Cycle (Ambler, 2005)

In practice, there is no single best development practice. Companies and project managers must select from the many available options based on their own requirements. Alternatively, project managers can study the various methods and select bits and pieces of each to develop a new development practice that fits their project. For example, Microsoft hires Program Managers that work with individual teams to decide how to build products. In doing so, they decide what software practices best fit their project's requirements. ("Microsoft university careers," 2013) There are many software development processes and many more sub-practices that make up these processes; because of this, a Software Process Configurator could be valuable to new project managers.

2.2 Decision Support Systems and Expert Systems

Decision Support Systems, or DSS, are computer-based systems that assist a person in solving problems based on the resources available to him or her. Unlike artificial intelligence tools, they are not

designed to automate decision-making, but rather provide suggestions and advice for human decision-makers. (Shim, 2002) For example, a decision support system might ask a user a series of questions relevant to what they might want to eat for dinner, and then give suggestions for what they should do. As the goal of the Software Profess Configurator is to give process recommendations to software project leaders, it is an example of a Decision Support System.

According to Introduction to Expert Systems, "An expert system is a computer program that represents and reasons with knowledge of some specialized subject with a view to solving problems or giving advice." (Jackson, 1999) The decision support system example above would be an expert system if food and psychology experts were to supply the system with the information to choose from. Likewise, the Software Process Configurator is an expert system as well as a decision support system.

Prior to the development of SPC 2.0, we researched DSS and Expert Systems with similar goals to the SPC, in order to determine what had been done in this area before. Among the most notable of our findings was the Software Development Practice Advisor, which can be found at http://www.fourth-medium.com/sdpa_demo.htm. Their product is described as follows: "The Software Development Practice Advisor is a patent-pending expert system that leverages Control Systems Engineering to enable the timely delivery of advice and guidance for software endeavors. It provides credible explanations as to why certain practices work in various contexts, and supports effective change through incremental adoption of practices in a culturally sensitive manner." (Fourth Medium Consulting, 2012)

2.3 Domain Specific Languages

According to Martin Fowler's Domain Specific Languages, a Domain Specific Language (DSL) is "a computer programming language of limited expressiveness focused on a particular domain." (Fowler & Parsons, 2011) In particular, he notes that it is a programming language much like Java and C++ are programming languages. The major difference between general-purpose languages (such as Java) and a DSL is its expressiveness. General-purpose languages provide more capabilities than a DSL, but are more difficult to learn and use as a result. On the other hand, DSLs are designed to be easy to learn, but are limited in usage.

The Software Process Configurator uses a DSL to describe Recommendations, which we will discuss in greater detail in the Methodology. This language was programmed using ANTLR, or "ANother Tool for Language Recognition." It is a parser generator that uses a formal language description called a

grammar to generate Java classes that read and parse the language. (Parr, 2012) The SPC also uses a simpler DSL to generate the results page.

2.4 The Software Process Configurator

The first version of the SPC, completed in March 2012, included a report much like this one, in which the previous group detailed areas that the project could be improved in the future. Because this version of the SPC was a proof of concept and not a full implementation, we also reviewed the code and class structure to determine what changes should be made in order to develop a fully functional system. For simplicity, we shall call the original project SPC 1.0, and this project SPC 2.0. The Unified Modeling Language (or UML) Diagram for the first version of the SPC is shown below.

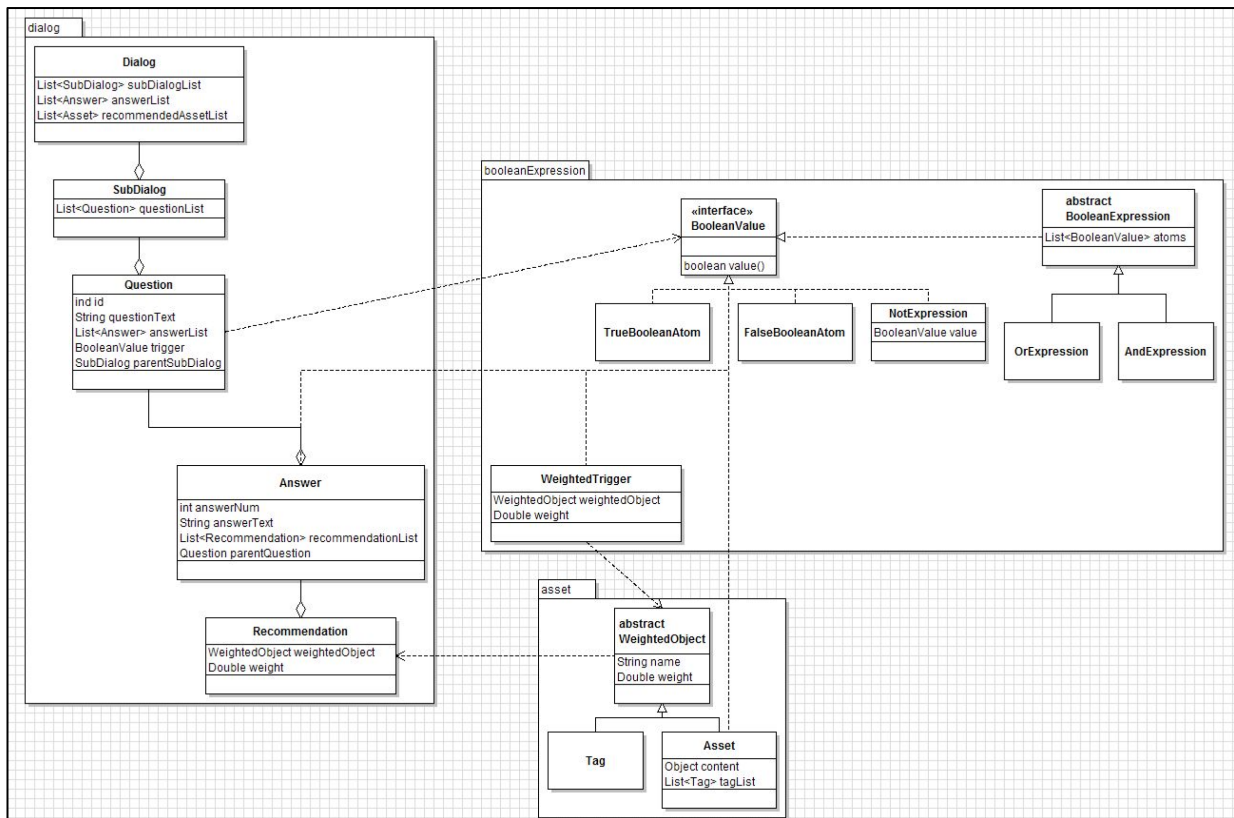


Figure 8: UML of SPC 1.0 (Todaro & Balasubramanian, 2012)

The most significant component of SPC 1.0 was the dialog system. Dialogs are written and edited in XML for a simple and effective means of developing short dialogs. This system is not quite suitable for a final product; not only is there no error-checking developed in the system, but it is not the simplest or most natural means for domain experts to convey their information. By focusing on providing the correct syntax, it becomes more difficult to provide questions for the dialogue. However,

it does provide flexibility for the system to develop. Because XML can be customized to add new tags and elements, it allows for changes in the underlying code structure. The dialog system is described in more detail in the Future Work section.

A major component of the SPC that was not completed in 1.0 is the Recommendations system. In SPC 1.0, full Recommendations did not exist. The results given to Process Selectors were Assets, and there was no database to store them. As a key function of the SPC is to be an expert system, it is essential that experts be able to access and modify this information. One of the major focuses of SPC 2.0 was to fix this system so that these Assets could be edited. We needed to change Assets to provide more information to users and to allow experts to edit them. In the next section, the Methodology, we explain in much more detail how and why we changed the system of recommendations, and the effects this had on the system as a whole.

An additional major lacking component of SPC 1.0 was a manager for experts to add their domain knowledge. The dialog was, and still is, edited in XML. Recommendations (or, at the time, Assets) were also specified in the dialog. We decided that this close coupling between the Recommendations and the dialog was not practical for the future, and a goal was to provide a system for Process Engineers to add information without having to use the dialog.

3.0 Methodology

With the first version of the Software Process Configurator (SPC), and our end goals of how the new system should behave, we set out to make SPC 2.0 a reality. This Methodology details how we evaluated the original system in terms of what needed to be changed, and what would need to be done in order to meet our objectives. It also describes the approaches we took in terms of completing work and organizing the project.

3.1 Requirements

As stated in the introduction, this project had three main objectives:

1. Allow users to provide relevant information regarding their software project,
2. Generate a report that recommends software development techniques, methods, and tools depending on the characteristics of their project, and
3. Provide resources that allow users to learn how to implement the suggested recommendations.

With these objectives as basic requirements, we selected specific goals for our project, and decided on the amount of work we could complete in the three seven-week terms given to us.

3.2 Scope

In order for our project scope to be feasible and still have practical applications, we decided to concentrate our system on the Software Engineering class taught by Professor Pollice, our advisor. In this class, students are split into teams and given a software engineering task to complete, simulating what might happen in a real software development company. The teams are small (around 15 people) and the experience and skill levels of students vary. This makes it a perfect target group for our application. The fact that it is a well-known class means that we can ask students for information about their experiences, and we can also use the next offering to test the application. Plus, the small teams and short time frame significantly limit the possible recommendations while still leaving enough to be useful. The varied skill levels will also give us information on how useful this tool is for people at both ends of the experience spectrum.

3.3 Goals

Throughout the entire project, two major goals guided our design. First, the SPC had to be flexible; we wanted out system to be easily expandable, modifiable, and adaptable. Although we are

using this system in the domain of software, we decided to design the system to be able to work for any domain. Second, we wanted it to be simple for people to get recommendations for their project, and also for domain experts to enter recommendations into the system. As important as flexibility is, making the system flexible should not sacrifice usability.

Besides these general goals, we also had specific goals for what we wanted our application to do by the end of the project timeframe. All these goals work towards our major goal of having the system ready to be tested by real students in the D term offering of Software Engineering. The following were our goals:

- Compile a useful set of recommendations to power the system
- Implement functionality to complete dialogs and get real recommendations
- Add an interface for experts to add and manage recommendations in the system
- Provide a customizable way to display results

In order to be robust and flexible, the Software Process Configurator needed to be organized into two main parts. The first part is a framework that allows for any decisioning process to occur. The second part is a collection of information that allows the SPC to provide informative and relevant suggestions. The final product should have the capability to perform other, unrelated decisioning processes if given a different set of information. It also allows for the framework to be developed without requiring the collection of information.

3.4 Evaluating Past Progress

To evaluate the existing system, we needed to understand how it was designed, how it works, and why certain decisions were made. The major resources we had to accomplish this goal were the code, the accompanying project report, and various manuals and references left behind by the previous group. Using all of these in tandem helped provide the understanding needed to decide what parts we could use and what parts we needed to replace.

The paper proved very useful with insights into the design decisions of the system, and the steps they went through to produce the SPC 1.0 codebase. It also contained diagrams and descriptions of sections of the code that greatly helped in the understanding of the entire system. The most useful part of the paper, however, was the “Future Work” section. This is where the authors explained where the current system falls short and what they would change in the future. In determining the state and usefulness of the current system this is exactly the information we needed.

The code extended the knowledge obtained from the paper, helping us put the pieces together and determine the exact state of the system. There were parts of the code that were both sensible and fully functional, and others that were not. There were classes that were implemented but never used in the system, showing where the authors were going but didn't have time to get to. It was evident how the design decisions described manifested in code.

After completing this analysis, we decided to use the existing system as a starting point to build upon. The dialog engine part of the system was implemented well, and was sufficient for our purposes. The part of the system that needed the most work was the recommendations system, which is exactly what we had wanted to expand upon. This means that by using the existing system, we could concentrate on the recommendations system and not have to re-implement a dialog system.

3.4.1 Problems with Old Design

The existing system was designed around the concept of a user going through a dialog, and thus the dialog was a central part of the system. The only way to specify recommendations was to provide them as part of an answer to a question. When you select that answer, the recommendation you provided gets added to the recommended list. This is problematic for both flexibility and usability. It strongly couples the dialog and the recommendations, meaning the recommendations cannot exist without the dialog. It also limits the way the system can pick recommendations, limiting the decision process to be based on a simple sequence of answers. Furthermore, we wished to give users an explanation of why each recommendation was chosen; coupling the dialog with the recommendation made this more difficult.

The presentation of the recommendations also had room for improvement. Using XML, Process Engineers had to write Results Templates to define what results they wanted displayed and how many. The results were given in plaintext at the end of the dialog. The first problem was that if a user wanted to change how the results looked, they had to understand how to edit the XML file. They might also have to implement Java code if the type of filter they wanted didn't exist. Next, the results were all displayed in plaintext. If the user wanted to display their results in some way – for example, on a web page – they would have to parse the output and put it into HTML. These were major issues in terms of usability.

Finally, in the existing system, recommendations and assets were indistinguishable, and as a result, recommendations lacked structure. This is another result of their dialog-centric design and really didn't fit into the recommendation-centric design we were intending to create. Because of all these

issues, we decided to design the recommendations system from scratch and then fit it in with the existing system.

3.5 Designing the Recommendation System

3.5.1 Decoupling from Dialog system

After deciding that the recommendations needed to exist on their own, independent from the dialog, we thought about what the structure of a recommendation should be and how a domain expert would want to describe a recommendation. The two major pieces were how a recommendation should be chosen and how it should be displayed to the user. The display would encompass the assets used to describe each recommendation, along with a description and summary of the recommendation. If the recommendations contained the logic for their own criteria for selection, it would allow them to be chosen in ways other than matching a single dialog question, and it would also provide an entry point for a system for adding new recommendations. Furthermore, having the recommendation keep track of how its chosen would allow for an explanation system for users to learn why each recommendation was chosen. These were all things we wanted to accomplish, so we decided to add a Recommendation interface with methods corresponding to each of the ideas discussed.

Another benefit of separating the recommendations from the dialog was that the domain expert does not have to write the dialog to add recommendations. Questionnaire and survey writing is a profession of its own, with specialists that would be better suited for writing the dialogs. Our design allows domain experts to focus on getting what they know into the system and lets someone else decide how the information should be gathered.

The next question we had to answer was how to connect independent recommendations with the existing dialog. For this we had to consider the purpose of the dialog. The dialog results in a list of answers, but that isn't what we care about. The important part is the information each answer represents, the project constraints we obtain from each answered question. This is also what each recommendation needs in order to decide whether it is a good fit. We came up with a system where, instead of building up answers, the dialog builds up constraints and gives the constraints to each recommendation so they can decide whether they should be chosen. Unlike an answer to a question, a constraint is independent of the information gathering process. For example, one constraint could be that the size of the project team is ten people. Constraints are specified as part of answers, giving each answer meaning in the system.

3.5.2 Implementing Recommendations

Our first step in implementing the Recommendation system was defining the interface for a recommendation. With an interface, future developers can make recommendation implementations that behave differently from ours. For example, this would allow someone to make a recommendation that decides whether it should be recommended using some sort of artificial intelligence. Each implementation of a Recommendation must provide the functions specified in the interface. These include functions for getting the name, description, summary, assets, and categories of the recommendation along with a function to evaluate a given set of constraints and return the confidence.

The next step was to implement the class that would represent these constraints. The dialog needed to be able to build up one of these constraint objects, and Recommendations had to use them for specification. We chose to represent individual constraints with a class called Attribute. An attribute has a name and a value representing the property of the project and its Integer, String, or Boolean value. For example, an attribute to represent the size of the team could have the name "teamSize" and a value of 4. We decided to store these attributes inside the Constraints object using a hash map with the names as the key and the entire attribute object as the value. This allowed for very quick adding and retrieval of attributes by name, which is the majority of what we need to do.

Next we needed to implement recommendations based on the interface we developed. We called our "basic" version of the recommendation interface a Basic Recommendation. The design of this class went through two different phases. We started with the philosophy that there would be a list of conditions that would describe when a recommendation should be recommended. Each condition was called a "Property." Each Property had an attribute that represented the condition which had to be true for the property to be satisfied. It also had a weight which was a number that represented how important the Property was. The weights of every satisfied property would be added up to get the recommendations confidence. Finally, a Property had a boolean indicating whether it was required or not. A recommendation would only be recommended if all its required properties were satisfied.

This approach worked for a while, but during our design of the DSL we realized that this design was not quite flexible enough. In particular, there could only be one scenario where a recommendation should be recommended. This lead to a new design; in this design, a recommendation could have multiple scenarios, each of which could affect the recommendation's confidence. To implement this, we created a new class called Scenario to represent one of these scenarios and removed the Property class. The Scenario class has a list of Attributes which behave similarly to the Attribute a Property had, defining the conditions where the Scenario applies. They also had a weight and a required flag, just like

properties. The difference is that these applied to the entire list of attributes, not just one. We also added the ability to add assets to a scenario, as well as the ability for a scenario to stop the recommendation from being recommended and stop the processing of scenarios all together. Assets that are attached to a scenario and not a recommendation are only part of the resulting recommendation if that scenario is satisfied.

At first we had the Recommendation's constraint evaluation function return an integer representing the confidence it should be recommended with. We quickly found that there were other things that the recommendation needed but that didn't exist until it was evaluated. The most notable example of this was the explanation of why the recommendation was recommended. At first, we simply added these things to the Basic Recommendation and stated in the documentation that they would only be available after the evaluation function was called, and that they pertained to the most recent of those calls. This had issues for many reasons. First, it could be confusing, and it is bad practice to have functions that only return values sometimes. Second, if we are saving recommendations in a database, we don't want to have to store their most recent result as well. And finally, if a function has a recommendation and passes it to another function, it cannot know if the recommendation results are changed by that function. We decided to create a class for recommendation results called RecommendationResult. This class holds the recommendation that was evaluated to get the result, as well as all the information that results from the evaluation including the confidence and the explanation.

3.5.3 Attributes and Values

The implementation of attribute values within the system took a considerable amount of thought and design. We needed each attribute to have a "value" where the value wasn't necessarily one value but could also be a range of values. Values also needed to have a type so they could be compared and collected through the dialog. As a starting point, we created an interface for what every value needed to have. Since we were using attributes both as constraints collected from the dialog and conditions for the scenarios, every value needed a function that took in another value and returned whether or not it fit the criteria of the attribute. For example, a value of 1 would fit the criteria of another value of 1 and also a range of 0-4, but not a value of 2 or a range of 2-4. Every value also needed a way to get its type, which we decided to represent as an enumeration. Our three basic types would be integer, string, and boolean. These were each useful in our application yet different enough that implementing them would help test our design.

The first design we came up with was too flexible to be practical. We decided to create a class hierarchy that would have two different subgroups of values: single and multiple. Single values

represented one primitive value, such as a single string, integer, or boolean. These were the building blocks of the multiple values. Multiple values represented a group of multiple single values. In our first version, the only multiple type we had was a range. We added a method to the interface to check the multiplicity of a value to reflect these two groups. For this design to work, the range had to take in another value class and work with all value types without having a special case for each. We accomplished this “arbitrary type” comparison by enforcing that all value classes must implement the comparable interface. This meant each single value controlled how it was compared to values of the same type, and all the range had to do was call compareTo. This allowed for single value classes to be independent of the multiple values, which made changing or adding either very easy.

The problem was that the code became confusing and difficult to maintain. This design allowed for a range to be of any value, including another range. Not only does a range of ranges not make any sense, but in order for a range to be comparable we needed to define how to compare a range with another range. This was nonsensical and at first we defined a consistent behavior because we only had one multiple-value class. However, when developing the DSL, we realized that we needed to represent open-ended ranges like less than and greater than, as well as a not value. The more multiple values we implemented, the more we realized that comparing multiple values didn’t make sense. Since our design hinged on every value being comparable, this meant we needed a better design.

In the final implementation of Values, we removed classes dedicated to specific types of multiple values and kept a single class per Value Type plus the ValueType enumeration itself. Each of the three types of Values (String, Integer, and Boolean) are implemented in a different way, thus the need for separate classes. For a Boolean, the class simply stores a Java boolean value, and isMultipleValue always returns false. No operator other than “equals” exists; greater than, less than, and between don’t make sense, a set of Booleans results in a pointless Value, and “not” can be expressed by the opposite value. Operators, including Single, Less Than, Greater Than, Between, and Not, are expressed in a ValueDataType enumeration.

Strings are more complicated, because it is possible for a Value to contain multiple strings. A String Value has three potential states besides empty: a single String, a set of multiple Strings, or everything except a single String. This is expressed by a list of Strings, and the function “equals” checks to see if there is a match in the contents of both Values. String Values are “built,” in that a String Value object is created, and then an “addString” function adds Strings to the Value.

Integer Values are the most complex, because these Values can be any combination of ValueDataTypes. Simple examples of Integer values include 1, both 5 and 7, everything less than 10,

everything between 0 and 4, and everything except 0. However, it is possible to represent something as complicated as $(-\infty,0]\cup 1\cup[3,6)\cup [8,20]\cup(50,\infty)$ with this implementation. An Integer Value stores a list of ValueData, and every ValueData contains a ValueDataType, a value, a second value when necessary, and an inclusive flag. This class can store one of any type of numerical value, and an Integer Value consists of one or many of these. With this specification, testing for multiplicity and adding information is simple. To determine if a Recommendation fits an Integer Value criterion, a fitting function checks for overlap between the two Integer Values by iterating through each ValueData in both Values and testing against an appropriate conditional statement each time.

As previously mentioned, the dialog needed to be modified to accumulate attributes instead of answers. Since the dialog is made using XML, this meant we needed to modify the XML to reflect this new change. We added an XML attribute called "attribute" to the question tag to specify the name of the attribute that the question was collecting. Next we created a new value tag to be placed inside answers. An answer can have multiple values, and values have both a type and a value XML attribute. The type is a string such as "lessThan" or "equal". We also added a shortcut XML attribute to the answer tag called "value" that allows you to specify an "equal" value without needing an inner value tag.

To support this in the parser, we needed to turn these string names and values into attribute objects. The first problem we faced was to determine the types of these attributes. We could infer the types using the values given, but this does not guarantee a match with the type of the attribute used in the recommendations. Attribute types should be standardized so the same attribute always has the same type, and there should be a way to look up that type. We implemented this using an AttributeReference class to store attributes and their descriptions. An AttributeDescription was another new class that held the name of the attribute, the type, and a string description of what the attribute represented. Inside the AttributeReference we stored these in a map with the attribute name as the key making it quick to lookup attribute descriptions by name. With this new standardized reference, the XML parser could get the attribute name, look up the type, convert the given value to that type, and create the attribute. If the user gave the wrong type, an exception is thrown.

3.5.4 Assets

We needed a way to represent generic assets in our system and database. These assets could be anything potentially helpful for a user to learn more about whatever they have been recommended. Possible Assets include websites, books, images, files, or even just plain text. For this project, we decided to only consider assets that could be stored as text. This is easy to implement with Db4o, and most Assets can be represented as a string in some way; for example, a file path could represent a file.

Our first design represented this arbitrary data using a map of strings to strings. The keys would be the names of the pieces of information, and the values would be the actual information. We added a function to return this map into an interface along with a function that would return the identifier or type of the asset. We thought every asset needed to store different information, so the best way was to simply have many implementations of the asset interface for the different types of assets. We would had a LinkAsset, a TextAsset, a OnlineImageAsset all as separate classes. But since they implemented asset, they all had to produce a map of their contents, which led to each class being implemented very similarly. In fact the only real difference was the keys of the map and the constructor. The duplicated code this created was undesirable and we decided to try and find a better way.

We still wanted the data to be represented as a map, but it was wasteful to have many classes each storing data in a map, so we did away with all the subclasses and created one generic subclass called AssetStore. In order to have different types of assets with their own consistent keys in the map, we created an enumeration for AssetType. This asset type not only defined what types of assets existed but also provided a way to get the content keys for each asset type. This preserved the ability to add an asset type without dependence on the other assets while removing the duplicated code. This new design caused some changes to the Asset interface; we replaced the string-based identifier function with one that used Assets, and we also added a function to get the Asset's content keys.

3.5.5 Presenting Recommendations

The next problem we encountered was figuring out how to present recommendations to the user. We needed a method that was both flexible, allowing recommendations to be presented how the user wants them, and usable, not requiring a large amount of customization just to get something. We decided upon a template system that would allow the user to take the information for each recommendation and put it in the form they wanted. We thought it was reasonable for users to format their results as HTML, JSON, or other common formats. We considered XML, since that is what the dialog uses and what the old system used, but it was too restricted. We wanted to provide the capability to format results as XML or similar languages, so using XML to express the results didn't work. We then looked at some of the template languages used in frameworks like Ruby on Rails and Django, and decided that they were similar to what we wanted but too complicated for both ourselves and for users. To use one of these already established templates engines would have required us to spend time integrating them into our project, and these languages are more powerful than necessary for the SPC. Our users would also have to learn whatever language we chose. So we chose to write our own very

small but powerful template language with a syntax that didn't overlap with any format we knew of. The resulting language was simple and allowed the results to be put into any desired plaintext format.

This template system was split into two parts: the recommendation template and the asset template. The recommendation template defines how to display each piece of the recommendation, including each asset. Then the asset template defines how each asset is displayed within the recommendation. This provides for maximum flexibility. Users can use one asset template per output language, and each can be used in many recommendation templates in order to display things differently.

3.6 Designing the DSL

3.6.1 How to Create Recommendations

A major addition we wanted to make to the previous system was to give experts a way to enter recommendations into the system, giving the system the knowledge it needs to be useful. But what was the most effective way to accomplish this? We wanted a method that was accessible, simple, and familiar to experts so they would actually write recommendations. We came up with two options: a Domain Specific Language (DSL) and a Graphical User Interface (GUI), both with strengths and weaknesses. A GUI would allow experts to build recommendations graphically using a custom user interface, selecting from options, typing in text boxes, etc. This type of interface is familiar, but it also has drawbacks. A DSL would allow experts to type up recommendations using a custom language. This may not be as intuitive to all experts, but for software experts this may not be the case.

Again, both options have strengths and weaknesses. In the GUI we can check errors as we go because we control the application, but with a DSL we can only check when the recommendation is loaded. On the other hand, a DSL can be written using any plaintext editor, some of which have features that an expert may be able to take advantage of. In a GUI, we would have to add any features we wanted ourselves. Furthermore, regardless of how we choose to implement it, it would be difficult to port. For example, if we wrote a desktop client and decided we wanted to change the SPC to be web based, we would have to rewrite the language. A DSL would use plaintext, making it easier to create a recommendation from anywhere. Both require a lot of upfront design to get the interface right. Based on these considerations, we chose to create a DSL. Flexibility was a primary concern, and the DSL option offered the most flexibility. It also still allows for a GUI to be designed in the future; in fact, it can run on top of the DSL. Finally, since our target is software experts, we thought using a programming language would be more familiar, faster, and more welcoming.

3.6.2 Designing the Syntax

A useful and useable DSL needs to have a syntax that fits its domain and makes it easier than a regular programming language. One of the first things we did was write out recommendations in Java using the objects that we designed early in the development process. To be effective, our DSL needed to be better than creating recommendations using native Java objects. Because of the class hierarchy used to create flexibility, constructing a Recommendation in Java involved multiple layers of objects within other objects, which is impractical if a Process Expert wishes to create many Recommendations. Besides providing confidence that our DSL would be much more efficient, this exercise also helped us visualize all the pieces of a recommendation we needed to represent. Each argument of the Recommendation constructor needed to appear in the DSL, but each argument was different and we needed to find a consistent representation. To help find a suitable representation, we explored DSLs that had been written for other systems.

While looking at many different DSLs for different applications, we came across Gherkin, the DSL used in Cucumber to define feature specifications. Cucumber is a tool for running automated acceptance tests. Stakeholders can write up the acceptance rules for a specific software feature (using Gherkin) and Cucumber can run this specification and tell the developers whether their code passes or not. Even though Gherkin is technically a DSL, it is very human-readable and accessible even to those who don't know programming languages. This is exactly what we wanted for our DSL, and we decided to use Gherkin as starting point. In Gherkin, each feature and piece of a feature you are describing is identified by a keyword followed by a colon. We decided our DSL would have the same syntax, each recommendation starting with "Recommendation:" followed by the name, and each section started similarly with a keyword and a colon. Since the description, summary, and category parts of recommendations are just text, they can be easily represented in this fashion and required no more specification. Following the same convention, different types of assets could be defined using the keyword "Asset" preceded by the type of asset; for example, "Link Asset". The different parts of the asset could then be defined in the same manner with the part name followed by a colon followed by the value; for example, "URL: www.google.com".

Next we needed to figure out how to represent Properties, which proved to be the hardest part. We started by again looking at Gherkin for some inspiration. Gherkin has something called a Scenario which describes a specific acceptance case for a feature. This sounded very similar to our Properties, which define when a recommendation is recommended. This comparison led to us envisioning Properties as Scenarios when a particular tool, process, or other item should be recommended. This

helped put into perspective what we wanted Properties to represent in the first place, which in turn led us to wonder why our current design only supported one scenario per recommendation. It is conceivable that there may be two or more very separate scenarios when something should be recommended. So, we decided to refactor Recommendations to have a list of Scenarios instead of having one list of Properties. We still weren't sure of everything the Scenario would contain, so we decided to design them in the DSL first and then find the best way to add them to the system.

```
Feature: Feedback when entering invalid credit card details

  In user testing we've seen a lot of people who made mistakes
  entering their credit card. We need to be as helpful as possible
  here to avoid losing users at this crucial stage of the
  transaction.

  Background:
    Given I have chosen some items to buy
    And I am about to enter my credit card details

  Scenario: Credit card number too short
    When I enter a card number that's only 15 digits long
    And all the other details are correct
    And I submit the form
    Then the form should be redisplayed
    And I should see a message advising me of the correct number of digits
```

Figure 9: A Gherkin Scenario (Wynne & Hellesoy, 2012)

Scenarios have two parts: the conditions, or the set of conditions determining whether a certain set of constraints qualifies, and the actions, or what to do when that set of conditions is true. At first there was only one possible action – to add confidence – but it was quickly evident that this was not sufficient. There could be scenarios when a recommendation should be avoided, and the expert might want to subtract confidence or force the confidence to 0. Oppositely, there could be a scenario when a recommendation works so well that should always be recommended. Finally, there could be a particular asset that helps only in a particular scenario, and it should be possibly to include these only when the project satisfies that scenario. Multiple actions can be specified by adding the word "And" followed by another action. Our final list of possible actions is:

- Add confidence
- Remove confidence
- Never recommend
- Always recommend
- Add asset
- Stop processing

The syntax for the conditions of the scenario needed to be human readable and powerful but not too complicated. We also didn't want it to be too verbose, making it tedious to type out. To maintain human readability, we decided conditions were best represented in the way they are spoken. If you were asked for a recommendation in person, you might say something like, "It works well given your team size is less than 4 and your timeframe is between 2 and 3 weeks." In terms of our recommendation system, team size and timeframe are the attribute names, less than and between are the operators, and the numbers are the values used by those operators. In our DSL, this statement looks almost the same: "Given team size is less than 4 And timeframe is between 2 and 3." All conditions are of the form "attribute name, operator, values" and you can combine them using "And" just as in the example. There is no "Or," because this would introduce a necessity for specifying order, adding unneeded complexity. An "Or" can be accomplished by simply making a new scenario with the same actions. Users also have the option to use the math versions of the operators instead of the English, or to drop the "is." In the previous example, you could write "is less than" as "<." The possible operators are:

- equals
- not equal
- less than, less than or equal to
- greater than, greater than or equal to
- between

You connect the conditions to the actions using the word "Then," making the full syntax as follows: "Scenario: Given <conditions> Then <actions>."

All the pieces described can be put into a recommendation in any order. The only sections that can appear multiple times are scenario and category. For the complete grammar for the DSL, see the Appendix.

```
Recommendation: Project Manager
    Category: SoftEngProject

    Summary:
    Team member with a dedicated Project Manager role

    Description:
    A Project Manager is a team member who is in charge of the team itself.
    He or she should organize meetings, provide agendas, schedule work, and help point
    the team in the right direction.

    Link Asset:
        URL: http://en.wikipedia.org/wiki/Project_manager
        Text: Project Manager - Wikipedia
```

Figure 10: A "Project Manager" Recommendation

3.6.3 Implementing the DSL

The next step was to implement the DSL so we could read in text-based Recommendations and produce new Recommendation objects in the system. Because our DSL ended up looking very similar to Gherkin, we first tried to see if we could use Gherkin and/or Cucumber to parse the DSL. The code for Gherkin can be found on GitHub. After reviewing it, we found that there was a large amount of code, it used a compiler tool we had never heard of, and it did not have a clear place for us to modify the grammar to fit our needs. To understand and adapt Gherkin would have taken more time then starting from scratch.

We explored many options of tools to aid in the implementation of the DSL. To satisfy the goals of a Major Qualifying Project, we wanted to spend our time implementing the DSL, not learning a new tool, so our options became using ANTLR and using regular expressions in Java. ANTLR is an LL* parser generator that, given a grammar for a language, can generate a custom parser. We started trying to implement our language with ANTLR because it seemed to be both the easiest and the most flexible place to start. Writing in ANTLR would be more readable, leave less room for mistakes, and be easier to change in the future.

This was a good idea in theory, but at first we had trouble because of how similar to natural language our DSL was. In a compiler, the lexer turns a stream of characters into a stream of tokens. The parser then goes through these tokens to create an Abstract Syntax Tree (AST). In past experiences with ANTLR it was easy to just ignore whitespace in the lexer, but in our language, whitespace could not be completely ignored due to the language including text-based descriptions and summaries. Whitespace needed to be captured, and because we found no way to partially ignore whitespace, we ended up dealing with it in the parser.

The next problem was to define the parts of the grammar that allow multi-line free text for specifying descriptions, summaries, etc. First we attempted to put this in the lexer so that in the parser, the entire text block could be one token. This did not work, because the lexer is less powerful than the parser; it cannot look ahead and backtrack in the same way. We did not figure this out right away, and we feared it was impossible to parse our grammar using ANTLR.

This setback led us to code up a simple Java-based parser to evaluate how hard it would be to implement the DSL with regular expressions. This worked well for the blocks of text, but it was clear that to parse our entire grammar this way would be tedious and unmaintainable. ANTLR was the better choice, and the implementation in Java gave us an idea of how to make it work. Instead of using the lexer to do the majority of the work, we delegated more difficult tasks to the parser. This worked much better, and by telling the parser to capture everything that wasn't a keyword followed by a colon, we were able to capture the blocks of text. The rest of the implementation of the parser was straightforward. The final step was to write a tree grammar to walk the AST and create the recommendation object.

As mentioned earlier, the parser turns a stream of tokens into an AST based on a set of rules. You can then write a tree grammar that will walk the AST and then generate code, check and evaluate variable types, check syntax, and much more. Our tree grammar needed to walk our AST and create a Recommendation object. To accomplish this, we wrote some extra utility classes to build each part of the recommendation from the pieces in the DSL. We could not just use the constructors for each object because the AST would not have all the pieces immediately. We could have modified the objects to have setters for every field, but this had the disadvantage of making those fields publically editable. Builders were the best option, holding each piece until every piece was gathered and they could construct the real objects. This worked very well and helped evaluate our design of the different objects that make up a recommendation. A simple and elegant builder is indicative of a good design, whereas a messy and confusing builder could help show where refactoring is needed. This helped us find problems with both our design of Assets and Values, which we then refactored.

3.7 Gathering Data

We have not yet touched upon one of the goals of our project: to provide real recommendations that Process Selectors can use. To do this, we had to fill the SPC database with Attributes and Recommendations that are applicable to Software Engineers, and we also had to write a dialog to determine if the right scenarios are met to give the recommendations.

Because we do not yet have a team of software process experts to fill the database with recommendations, we instead sent a survey to students who had taken Professor Pollice's Software Engineering class. This survey can be found in the Appendix. By analyzing the results of this survey, we determined what development methods were used by these teams, and the scenarios for which the methods were useful. With this information, we added some sample recommendations and attributes to the database. Some of the recommendations we added can also be found in the Appendix, and they serve as an example of how the DSL for Recommendations is used in practice. We then used the Attributes to help write a Dialog for users to go through. Parts of this dialog are shown in the Use Cases in the Results Section, as is the process of adding recommendations and attributes to the database.

4.0 Results

The goal of this project was to produce a fully functional tool (the Software Process Configurator) that would be able to:

1. Allow users to input information about their software project,
2. Generate a report with recommendations based on their project, and
3. Provide resources for users to better apply these recommendations.

The first version of the SPC, completed in 2012, was a proof of concept that primarily focused on solving the first of these three objectives. This second version is capable of providing the project-specific recommendations as well as related resources. While it is not yet a finished product, SPC 2.0 can finally provide users with the tools they need to begin their projects. In this section, we describe how a Process Engineer might use the SPC to add their recommendations, and how a Process Selector would go about using the system. For a complete specification of the SPC, see the UML diagram in the Appendix.

This section contains two Use Cases, one for a Process Engineer, and one for a Process Selector. The first use case describes how an expert might go about adding a Recommendation to the system. The second use case shows how the added Recommendation would affect a user's experience while using the SPC.

4.1 Use Case – Process Engineer

Gary is the professor of a software engineering class and has been for many years. His class involves students working in large groups to develop real software in a short period of time. He knows what works and what doesn't for different kinds of teams from both his industry experience and his experience with the class. He has decided to provide his students with a dialog for the SPC in order to help them in the class. In order for the SPC to be useful, he needs to turn his knowledge and advice into recommendations for the SPC. One thing he knows is that teams with little experience should meet often if they have time. He also knows that meeting too often can be bad for teams with a lot of experience since they get distracted and aren't as productive. If the team can meet every day he would also like to give them information about daily stand up meetings.

Using the SPC DSL, he turns all this information into one concise recommendation. He starts by giving this recommendation a name, "Meet Often As An Entire Group." He does this by opening up his favorite text editor and typing:

```
Recommendation: Meet Often As An Entire Group
```

He then gives the recommendation a quick summary of what it means, and a longer description that provides more detail. He types:

```
Summary: As often as possible meet as an entire group.  This helps
everyone know what is going on and help each other out.
Description:
      It may seem like meeting as an entire group many times a week
would be a waste of time, but it can actually be very helpful.  Group
meetings can be used for planning the week, but they can also be
working meetings where everyone can work in the same room and get help
and clarification when needed.  This eliminates wasted time when
people get stuck or are not sure what they need to do.
```

Gary has also asked past classes for their advice to future classes. Quite a few of the responses have to do with meeting often, so he wants to point students towards the course webpage where these testimonials are. He does this by adding an asset to the recommendation. He types:

```
Link Asset:
      url: http://web.cs.wpi.edu/~gpollice/testimonials.html
      text: Testimonials from Past Students
```

Now Gary needs to add the scenarios for when this should be recommended. The first scenario is that if a team is inexperienced and can meet often, then they should. The two pieces of information he needs are how experienced the team is and how often they can meet per week. He opens the Recommendation Tool to see if there are any attributes already in the system for what he needs. He finds one attribute called "team_experience" that is an integer that is described as "the average team experience as a rating from 1-10." He will use this to check the experience but will have to create a new one for how often teams can meet per a week. To add this to his recommendation, he types:

```
Scenario:
      Given team_experience <= 5 And
             meeting_days_per_week > 3
      Then Add 100 confidence
```

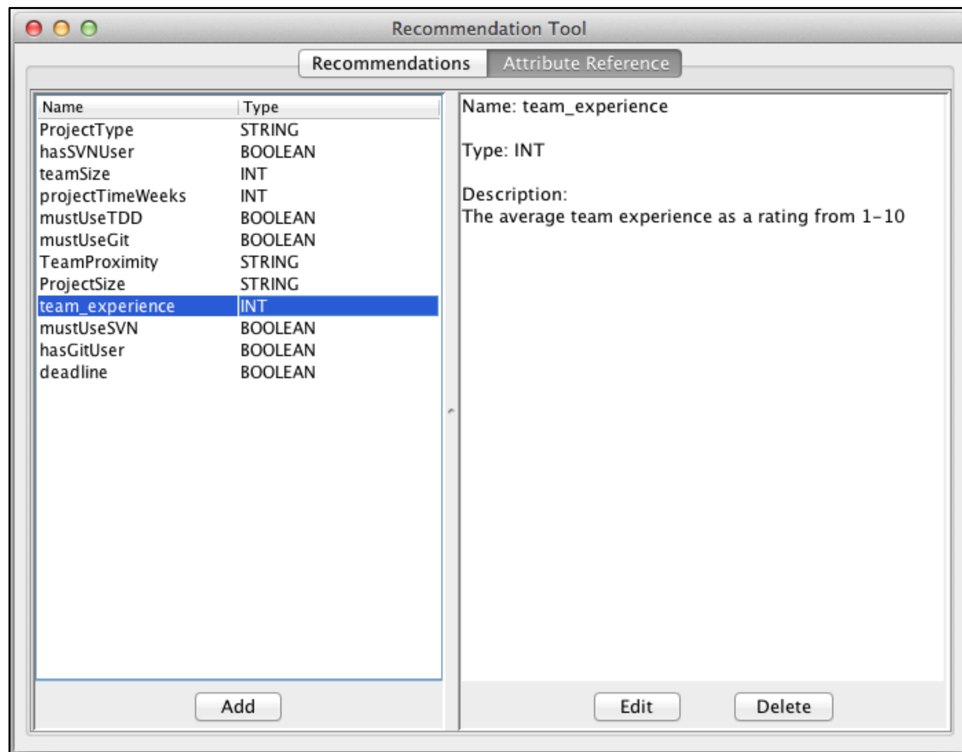


Figure 11: The Recommendation Tool for Domain Experts

He adds a confidence of 100 because he wants this scenario to cause this recommendation to be recommended, but he also wants other scenarios to be able to subtract confidence in case they meet negative criteria. The next scenario describes the case where teams with a lot of experience shouldn't meet as often. He types:

```
Scenario:
    Given team_experience > 8
    Then Remove 50 Confidence
```

Finally he wants to add an extra link to information about daily standup meetings for groups that can meet every day. He can do this by creating a scenario that adds an asset. He types:

```
Scenario:
    Given meeting_days_per_week >= 5
    Then Add { Link Asset:
        url: http://en.wikipedia.org/wiki/Stand-up_meeting
        text: Try a daily standup meeting }
```

```
Recommendation: Meet Often As An Entire Group

Summary: As often as possible meet as an entire group. This helps everyone know what
is going on and help each other out.

Description:
    It may seem like meeting as an entire group many times a week would be a waste of
     time, but it can actually be very helpful. Group meetings can be used for
    planning the week, but they can also be working meetings where everyone can work
    in the same room and get help and clarification when needed. This eliminates
    wasted time when people get stuck or are not sure what they need to do.

Link Asset:
    url: http://web.cs.wpi.edu/~gpollice/testimonials.html
    text: Testimonials from Past Students

Scenario:
    Given team_experience <= 5 And
        meeting_days_per_week > 3
    Then Add 100 confidence

Scenario:
    Given team_experience > 8
    Then Remove 50 Confidence

Scenario:
    Given meeting_days_per_week >= 5
    Then Add { Link Asset:
        url: http://en.wikipedia.org/wiki/Stand-up_meeting
        text: Try a daily standup meeting }

```

Figure 12: Sample Recommendation

This completes the recommendation and now he adds it to the SPC database. He opens up the Recommendation Tool that comes with the SPC. First he adds the attributes he used in his recommendation to the Attribute Reference. He adds meeting_days_per_week as an integer attribute and gives it the description "the number of days an entire team can meet each week."

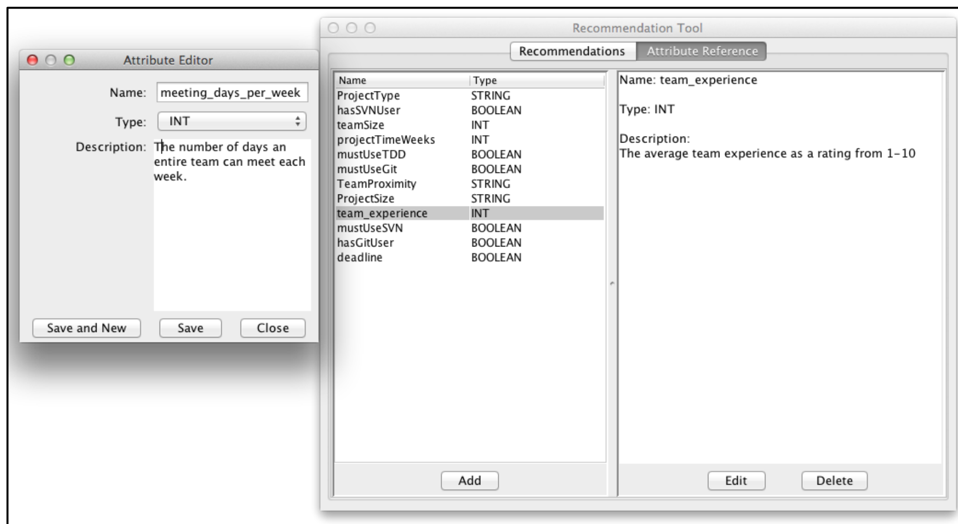


Figure 13: Adding a new Attribute

Now he switches to the Recommendation tab and loads in his new recommendation by clicking Add and finding the recommendation on his computer.

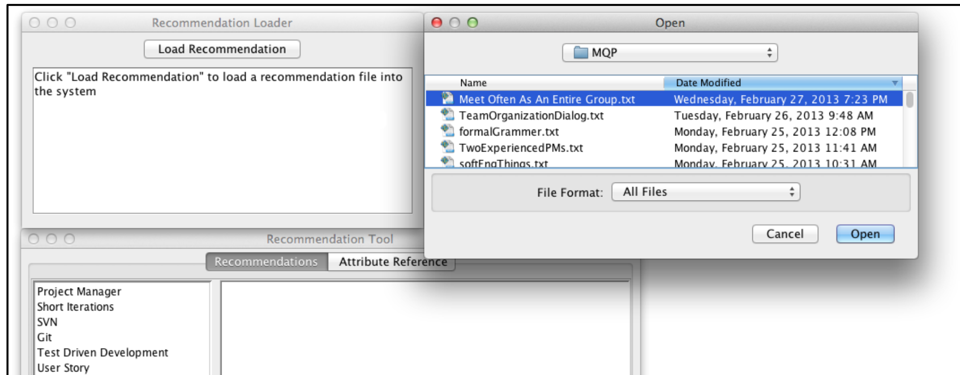


Figure 14: Finding a Recommendation to add

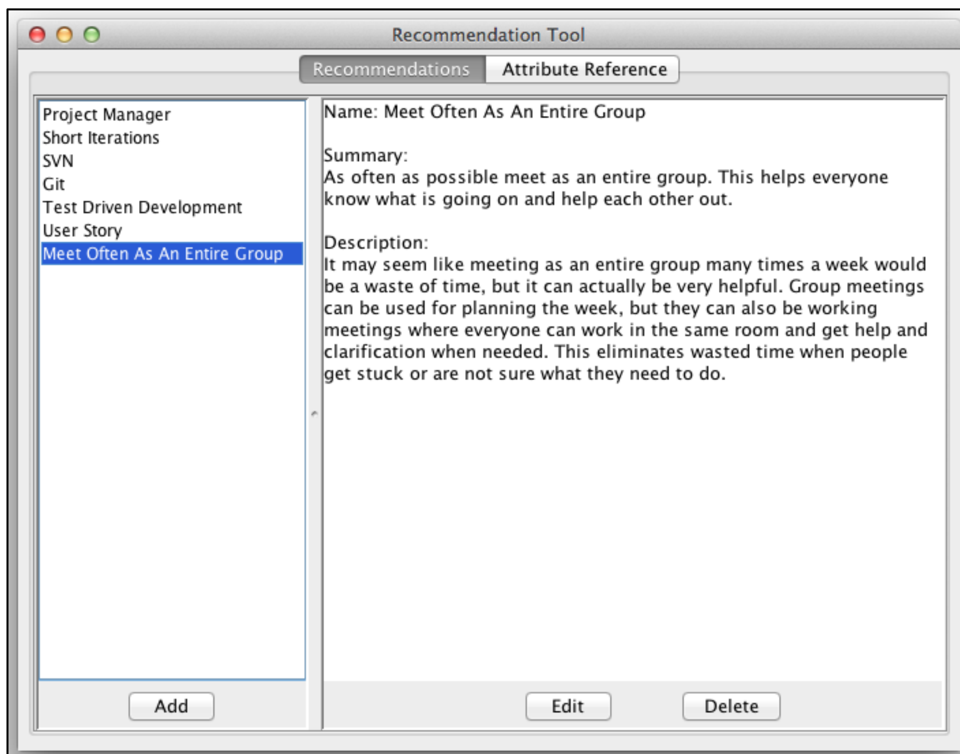


Figure 15: The Recommendation is loaded into the system

Gary follows the same basic steps and creates other recommendations, including using iterations of length 1 week, using pair programming when possible, and using the two most experienced project managers in the first and last weeks. Now he needs to write a dialog so students can input their specific information and get the appropriate recommendations. For his frequent meeting

recommendation he needs to ask two questions and make sure they populate the appropriate attributes. Inside his dialog XML he creates these two questions as follows:

```
<Question id=”1” questionStatement=”How many days a week can your entire team meet?” attribute=”meeting_days_per_week”>
        <Answer text=”1-2” value=”1”>
                <Value type=”between” start=”1” end=”2”>
</Answer>
        <Answer text=”3” value=”3”></Answer>
<Answer text=”4” value=”4”></Answer>
<Answer text=”5” value=”5”></Answer>
</Question>
<Question id=”2” questionStatement=”On a scale from 1-10 what is the average programing experience of your team?” attribute=”team_experience”>
        <Answer text=”1” value=”1”></Answer>
<Answer text=”2” value=”2”></Answer>
        ...
</Question>
```

```
<Question id="1" questionStatement="How many days a week can your entire team
meet?" attribute="meeting_days_per_week">
    <Answer text="1-2" value="1">
        <Value type="between" start="1" end="2">
    </Answer>
    <Answer text="3" value="3"></Answer>
    <Answer text="4" value="4"></Answer>
    <Answer text="5" value="5"></Answer>
</Question>

<Question id="2" questionStatement="On a scale from 1-10 what is the average
programing experience of your team?" attribute="team_experience">
    <Answer text="1" value="1"></Answer>
    <Answer text="2" value="2"></Answer>
    <Answer text="3" value="3"></Answer>
    <Answer text="4" value="4"></Answer>
    <Answer text="5" value="5"></Answer>
    <Answer text="6" value="6"></Answer>
    <Answer text="7" value="7"></Answer>
    <Answer text="8" value="8"></Answer>
    <Answer text="9" value="9"></Answer>
    <Answer text="10" value="10"></Answer>
</Question>
```

Figure 16: Sample Dialog Information

With recommendations in the system and a dialog, Gary can now give the SPC to students and allow them to get recommendations for how to complete their project.

4.2 Use Case – Process Selector

Jack is a student in a software engineering class where they need to complete a real software project during the length of one class. He wants to do well in the class and decides to make sure his team is prepared to produce the best software. He decided to use the SPC dialog created for his class in order to get some recommendations about how his team should operate and what tools they should use. The SPC asks the following questions and he provides the following answers:

Q: How long do you have to complete the project?

A: 7 weeks

Q: Does anyone on your team have project management experience?

A: Yes

Q: How many people?

A: 2

Q: How many days a week can your entire team meet?

A: 4

Q: On a scale from 1-10 what is the average programing experience of your team?

A: 5

Q: How many people on your team would you be rated above a 7 for programming experience?

A: 2

The SPC then recommends for Jacks team to use short iterations of 1 week, meet as often as possible as an entire team, to use pair programming when possible, and to use the two experience project managers in the first and last weeks. It provides resources to learn more about each of these practices. It also explains why each one is good for their project. For example, it says that the first and last iterations are the most important and require the most leadership, which is why you should use your most experienced project managers during those iterations.

5.0 Future Work

The Software Process Configurator 2.0 is by no means a final product. We have referred to the first version of the SPC as a proof of concept. The second version could be considered an Alpha version; in other words, it is feature complete. The third version of the SPC should move from the Alpha version to a Beta version that can be distributed to testers and Process Engineers. The advantage of this would be that the database would develop a larger collection of expert knowledge.

In order to move from this Alpha version to a fully functional prototype, we have compiled some suggestions for future work. The highest priority items are to implement a client-server database, to construct a tool for creating dialogs, and to add more information to the database. We will also suggest additional improvements later in this section.

5.1 Client-Server Database

The Software Process Configurator, as stated in the Background, is an Expert System. That is, it uses a collection of information from experts to make decisions. However, in order for experts to add this information, there must be a place to store it. SPC 2.0 has a DB4O database for this purpose, but it is a client-side database that comes with the SPC itself. This is not conducive to an expert system, because if there are multiple experts who wish to add information, they would need to use a single database file shared between them.

To fix this problem, we recommend implementing a client-server system for the database. If the database is stored on a server, then clients using the SPC could connect to it instead of using a local copy of the database. Furthermore, if the database is stored on a server, then it may be possible to make the entire application web-based. We recommend that any future groups look into this option to see if it is feasible.

5.2 Dialog Tool

One of the largest improvements between SPC 1.0 and SPC 2.0 is the Recommendation system. Experts can use the DSL we provide to write Recommendations of their own, and can use the tool we provide to add, delete, and modify Recommendation in the database. This tool also allows experts to view and edit the Attributes in the system.

In the next Software Process Configurator, we recommend to further improve this tool. Version 2.0 has no user interface for editing the dialogs that users interact with. If an expert wants to modify a dialog, he or she will have to edit the XML directly. SPC 3.0 should include a tool for experts to modify

the dialog in addition to attributes and recommendations. It could either be another domain specific language or a GUI editor.

5.3 Expert Knowledge

The SPC relies on expert knowledge to provide recommendations to users. We added a DSL to make it easier for experts to turn their knowledge into something that is useful for the system, but experts still have to write these recommendations to make the system useful. We used the experiences of students who have taken Software Engineering to create some recommendations for that specific use, but that can only go so far. To make the SPC useful outside of this Software Engineering class, future groups need to add more recommendations to the system. The easiest way to do this would be to get the system into the hands of many different kinds of experts and have them build up the knowledge base on their own. This would require some of the other improvements we have mentioned earlier, but it could turn the SPC into something that developers of all kinds could use in order to help figure out where to start.

Short of getting real experts to write recommendations, future groups could continue to grow the knowledge base by collecting more data and getting professors to contribute. WPI professors are certainly experts in their field and might be more willing than others to contribute to the system. Future teams could gather more information outside of just the software engineering class. They could ask students about how they decided to start personal projects or how companies they have worked for have operated and why. This would allow the creation of more recommendations outside of the scope we decided on.

5.4 Web System

Turning the SPC into a web application would change it from a feature complete Alpha to a usable and useful Beta. The system that has been built so far could in theory be converted to a web application. The backend could be written in Java and use the same code and Db4o database that the current system uses. The front end would have to be converted into HTML and Javascript, but this is a very small part of the system that could use improvement anyway. A user and permission system could be set up to control who could add recommendations, edit attributes, and use the dialog. A web system would also require no downloads or installation and would make the system more usable for everyone. If a server is dedicated to the SPC, this would be a great location for a centralized database, and there could even be a web based recommendation editor that helps experts write recommendations in the DSL.

5.5 Improved Dialog Options

The dialog system that was given to us as part of the past MQP was sufficient for our needs and gave a good base to start with; however, in creating a dialog for software engineering students, we realized it was quite limited. Currently there is only one type of question: a multiple choice question where only one answer can be selected. Other types of questions would be very useful in creating an efficient and user friendly dialog. Looking at popular survey software like SurveyMonkey and Google Docs, you can get examples of many different types of useful questions. These include multiple selects, grids, short answers, and drop down lists. While writing our dialog, we wanted students to be able to enter the languages their group was familiar with, and how familiar they were with each. The only way to accomplish this was to create a list of languages and ask a question for each. With a grid, or even a multiple select, this could be more efficient. In general, different types of questions will streamline and shorten dialogs.

5.6 Improved Explanation System

One of the things we designed but didn't have time to completely implement was the explanation system. The explanation system is the part of an expert system that can justify to the user why something was chosen. Our current explanation system simply says what scenarios were satisfied without giving any other information. In our property-based design this explanation was more useful, but when we moved to scenarios this value was lost because we had no brief way to describe a scenario. An improved explanation system would be much more helpful and might be more complicated than a simple explanation string. It might be useful after a recommendation to query the explanation system to see how one specific attribute affected the confidence. A decision diagram of how the recommendation was evaluated could also be useful, but may be more complicated than necessary.

6.0 References

Ambler, S. W. (2005). FDD life cycle. Retrieved from http://www.agilemodeling.com/images/lifecycleFDD.gif

Beck, K. et al (2001). Manifesto for Agile software development. Retrieved from http://agilemanifesto.org/

Boehm, B. (1986). A spiral model of software development and enhancement. ACM SIGSOFT Software Engineering Notes, 11(4), 14-24. doi: 10.1145/12944.12948

Fourth Medium Consulting Inc., (2012). Retrieved from website: http://www.fourth-medium.com/sdpa_demo.htm

Fowler, M., & Parsons, R. (2011). Domain-specific languages. Addison-Wesley.

IBM Corporation, (2007). IBM Rational Unified Process. Retrieved from website: ftp://public.dhe.ibm.com/software/rational/web/datasheets/RUP_DS.pdf

Jackson, P. (1999). Introduction to expert systems. (3 ed.). Harlow, England: Pearson Education Limited.

Kennaley, M. (2010). SDLC 3.0: Beyond a tacit understanding of Agile. (pp. 24-36). Fourth Medium Press.

Lacey, M. (2012). Scrum framework flow. Retrieved from http://www.scrumalliance.org/system/resource_files/0000/3905/Scrum_Framework_Flow.png

May, E. L., & Zimmer, B. A. (1996, August). The evolutionary development model for software. Retrieved from http://www.hpl.hp.com/hpjournal/96aug/aug96a4.pdf

Melonfire. (2006, September 22). Understanding the pros and cons of the waterfall model of software development. Retrieved from http://www.techrepublic.com/article/understanding-the-pros-and-cons-of-the-waterfall-model-of-software-development/6118423

Microsoft Corporation, (2013). Microsoft university careers - software development jobs. Retrieved from website: http://careers.microsoft.com/careers/en/us/grad-software-jobs.aspx

Naur, P., & Randell, B. (1968, October). Software engineering, NATO science committee. Retrieved from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

Parr, T. (2012). ANTLR. Retrieved from http://www.antlr.org/index.html

Rerych, M. (2002). Waterfall model - creation context. Retrieved from https://cartoon.iguw.tuwien.ac.at/fit/fit01/wasserfall/entstehung.html

Shim, J. P. et al (2002). Directions for the next decade. Decision Support Systems, 33(2), 111-126. doi: 10.1016/S0167-9236(01)00139-7

Todaro, J., & Balasubramanian, P. (2012). Software process configurator. Unpublished manuscript, Computer Science, Worcester Polytechnic Institute, Worcester, MA, .

U.S. Department of Labor, Bureau of Labor Statistics. (2013). Occupational employment statistics. Retrieved from website: http://www.bls.gov/oes/tables.htm

Wynne, M., & Hellesoy, A. (2012). The cucumber book: Behaviour-driven development for testers and developers.

XP practices. (2008, June 3). Retrieved from http://xprogramming.com/images/circles.jpg

Appendix A Glossary of Terms

The following terms are used throughout this report, and are defined here for reference.

Asset: A resource for a particular Recommendation that allows a user to act upon the Recommendation given to them. It could be a book or a website that gives more information, or even a tool to help implement the suggestion.

Attribute: Information about the project that a user might specify. A set of Attributes can describe any given software project. Attributes contain values and names. Attributes describing a project are compared with the Attributes of a Recommendation's Scenarios.

DSL: Domain Specific Language. A DSL is a programming language that is developed for specific uses, as opposed to multi-purpose languages such as Java. In this report, DSL often refers to the language used to specify a Recommendation.

Process Engineer: A Domain Expert in the field of Software Engineering with expertise on software processes. The Process Engineer provides Recommendations to the SPC.

Process Selector: The end-user for the SPC; the person who is looking for software development advice. The Process Selector uses the dialogue to provide information about their project.

Recommendation: A specific set of information provided by a Process Engineer and given to a Process Selector after using the SPC. It contains properties that describe the recommendation internally, references that are used to help instruct the user, and a description that allows a user to understand what is being recommended.

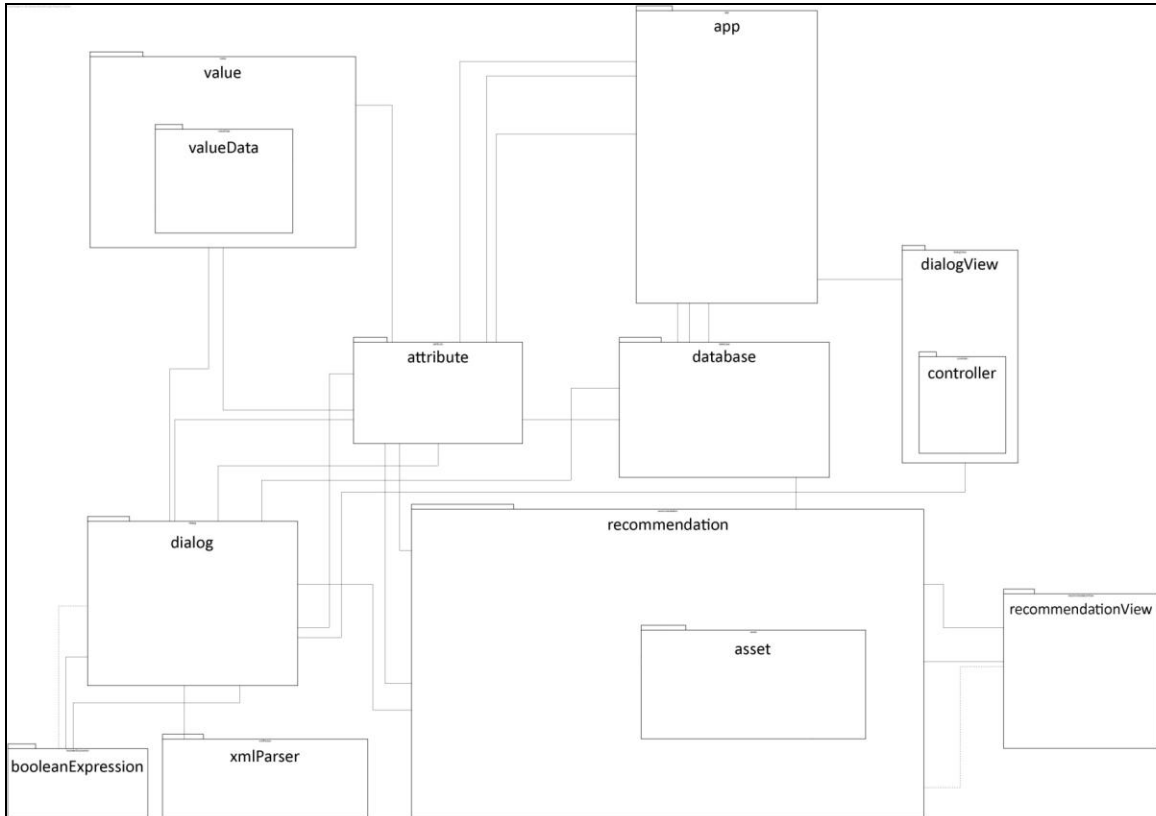
Scenario: A description of a situation for which a Recommendation might be given. Each Scenario depends on a set of Attributes, and actions are taken if the conditions are met. These actions include adding confidence for the Recommendation, making it a requirement, or adding specific Assets.

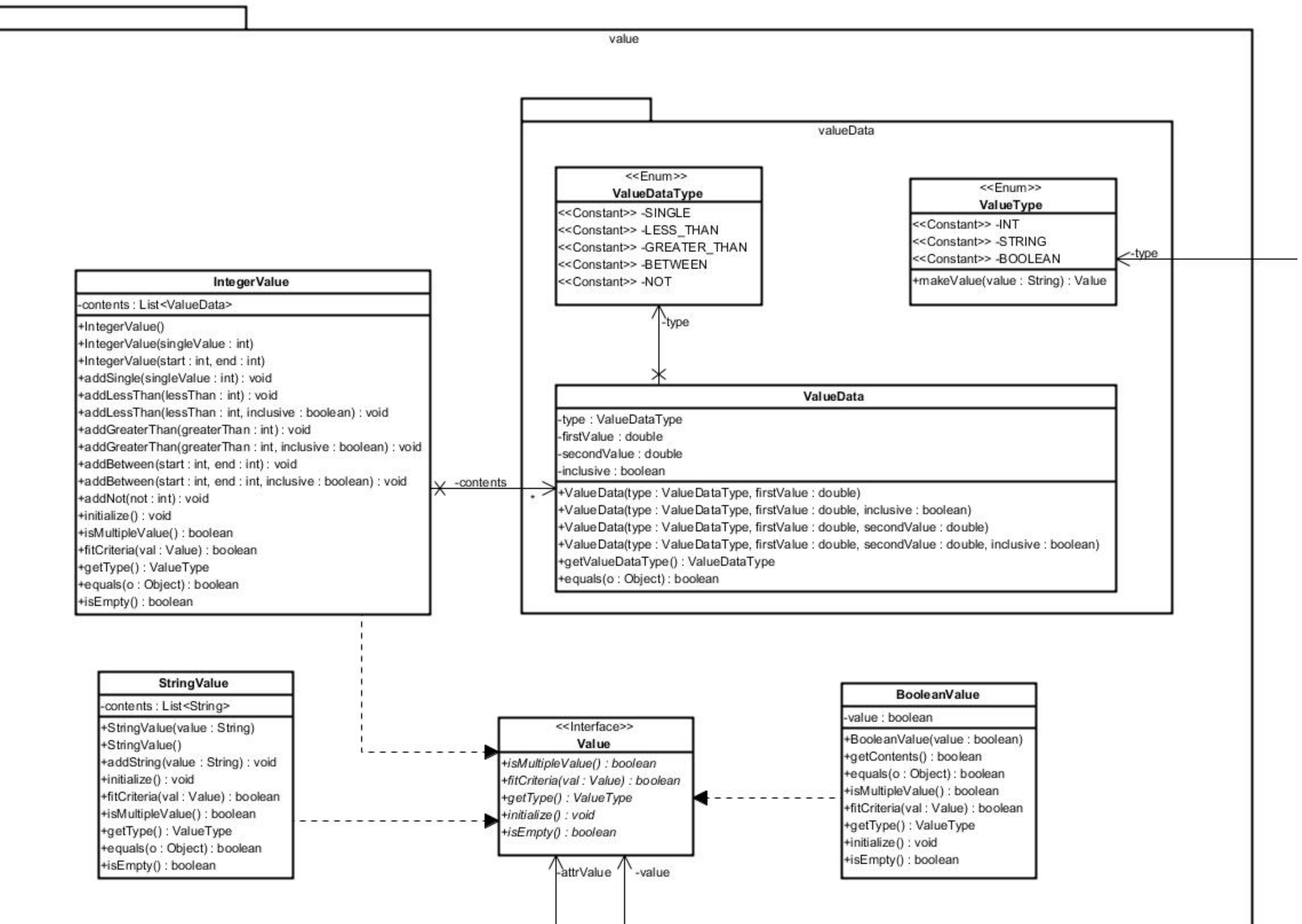
SPC: Software Process Configurator, or SPC 2.0. The tool used for providing expert assistance in software development.

SPC 1.0: The first version of the Software Process Configuration completed in March 2012 by Pragathi Balasubramanian and Jake Todaro.

Appendix B SPC 2.0 Unified Modeling Language (UML) Diagram

The following diagram is a high-level description of the packages in the SPC system. The next pages offer close-up UML views of individual packages. The full UML can be found in the files attached to this report.




value
valueData
<<Enum>>
ValueDataType
<<Constant>> -SINGLE
<<Constant>> -LESS_THAN
<<Constant>> -GREATER_THAN
<<Constant>> -BETWEEN
<<Constant>> -NOT
<<Enum>>
ValueType
<<Constant>> -INT
<<Constant>> -STRING
<<Constant>> -BOOLEAN
+makeValue(value : String) : Value
-type
-type
ValueData
-type : ValueDataType
-firstValue : double
-secondValue : double
-inclusive : boolean
+ValueData(type : ValueDataType, firstValue : double)
+ValueData(type : ValueDataType, firstValue : double, inclusive : boolean)
+ValueData(type : ValueDataType, firstValue : double, secondValue : double)
+ValueData(type : ValueDataType, firstValue : double, secondValue : double, inclusive : boolean)
+getValueDataType() : ValueDataType
+equals(o : Object) : boolean
IntegerValue
-contents : List<ValueData>
+IntegerValue()
+IntegerValue(singleValue : int)
+IntegerValue(start : int, end : int)
+addSingle(singleValue : int) : void
+addLessThan(lessThan : int) : void
+addLessThan(lessThan : int, inclusive : boolean) : void
+addGreaterThan(greaterThan : int) : void
+addGreaterThan(greaterThan : int, inclusive : boolean) : void
+addBetween(start : int, end : int) : void
+addBetween(start : int, end : int, inclusive : boolean) : void
+addNot(not : int) : void
+initialize() : void
+isMultipleValue() : boolean
+fitCriteria(val : Value) : boolean
+getType() : ValueType
+equals(o : Object) : boolean
+isEmpty() : boolean
-contents
*
StringValue
-contents : List<String>
+StringValue(value : String)
+StringValue()
+addString(value : String) : void
+initialize() : void
+fitCriteria(val : Value) : boolean
+isMultipleValue() : boolean
+getType() : ValueType
+equals(o : Object) : boolean
+isEmpty() : boolean
<<Interface>>
Value
+isMultipleValue() : boolean
+fitCriteria(val : Value) : boolean
+getType() : ValueType
+initialize() : void
+isEmpty() : boolean
-attrValue
-value
BooleanValue
-value : boolean
+BooleanValue(value : boolean)
+getContents() : boolean
+equals(o : Object) : boolean
+isMultipleValue() : boolean
+fitCriteria(val : Value) : boolean
+getType() : ValueType
+initialize() : void
+isEmpty() : boolean

attribute

-attr

AttributeDescription

-name : String
-type : ValueType
-description : String

+AttributeDescription(name : String, type : ValueType, description : String)
+AttributeDescription(name : String, type : ValueType)
+makeAttribute(value : Value) : Attribute

-attrDesc

-attrRef -attrRef

AttributeReference

-availableAttrs : Map<String, AttributeDescription>

+AttributeReference()
+getAttr(name : String) : AttributeDescription
+getAllAvaiableAttrs() : Collection<AttributeDescription>
+searchForAttr(search : String) : Collection<AttributeDescription>
+addAttribute(attrDesc : AttributeDescription) : void
+removeAttribute(name : String) : void

-attrRef

Attribute

-name : String
-value : Value

+Attribute(name : String, value : Value)
+equals(o : Object) : boolean

-attr

-attribute -attribute -attrRef

database

-db -db -db

<<Interface>>
Database

+use(dbName : String) : void
+store(o : Object) : boolean
+delete(o : Object) : boolean
+close() : void
+getAllRecommendations() : Collection<Recommendation>
+getAttributeReference() : AttributeReference
+getRecommendation(name : String) : Recommendation

-database

Db4oDatabase

-dboc : ObjectContainer

+Db4oDatabase()
+use(dbName : String) : void
+close() : void
+store(o : Object) : boolean
+delete(o : Object) : boolean
+getAllRecommendations() : Collection<Recommendation>
+getAttributeReference() : AttributeReference
+getRecommendation(name : String) : Recommendation

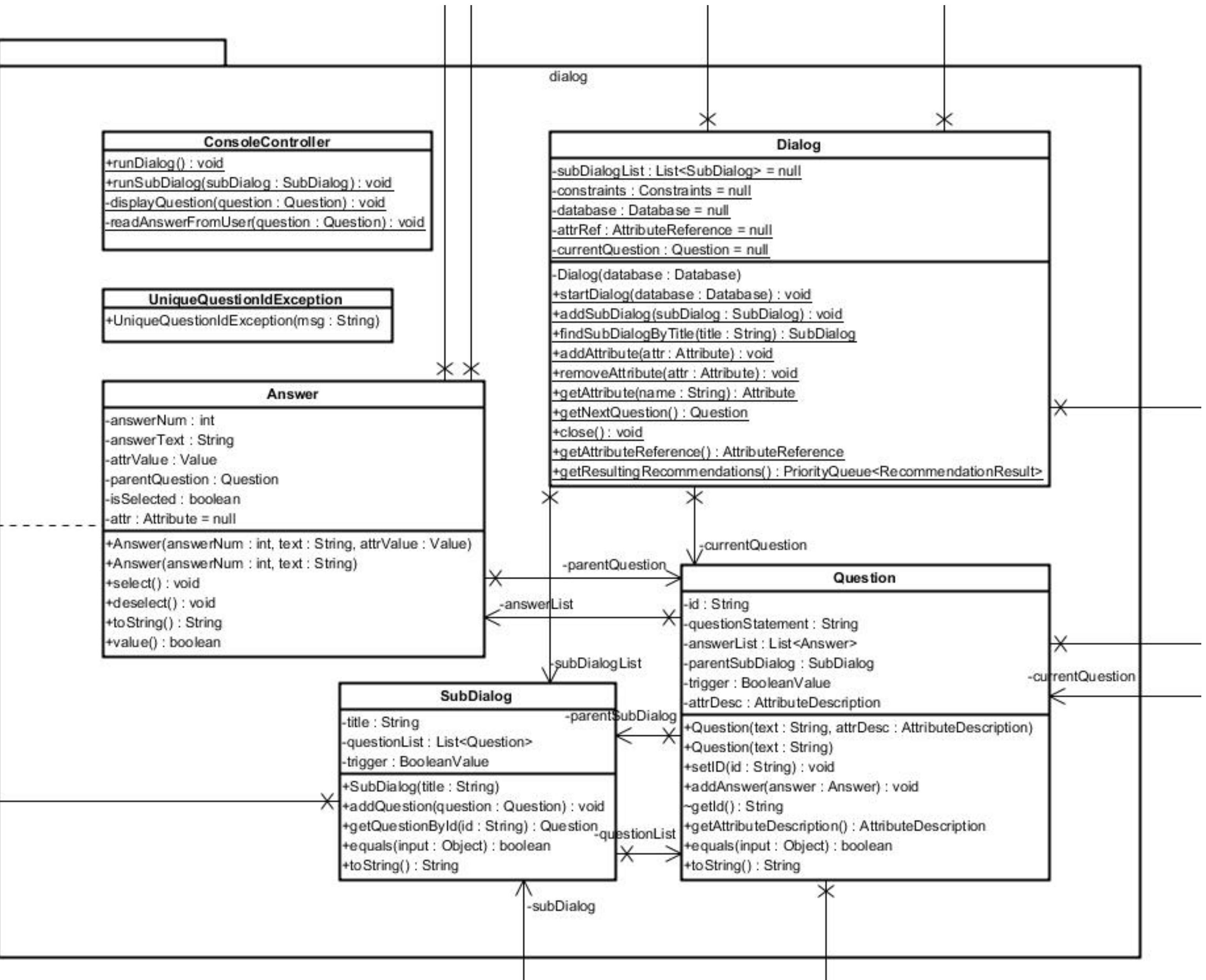
TestDatabase

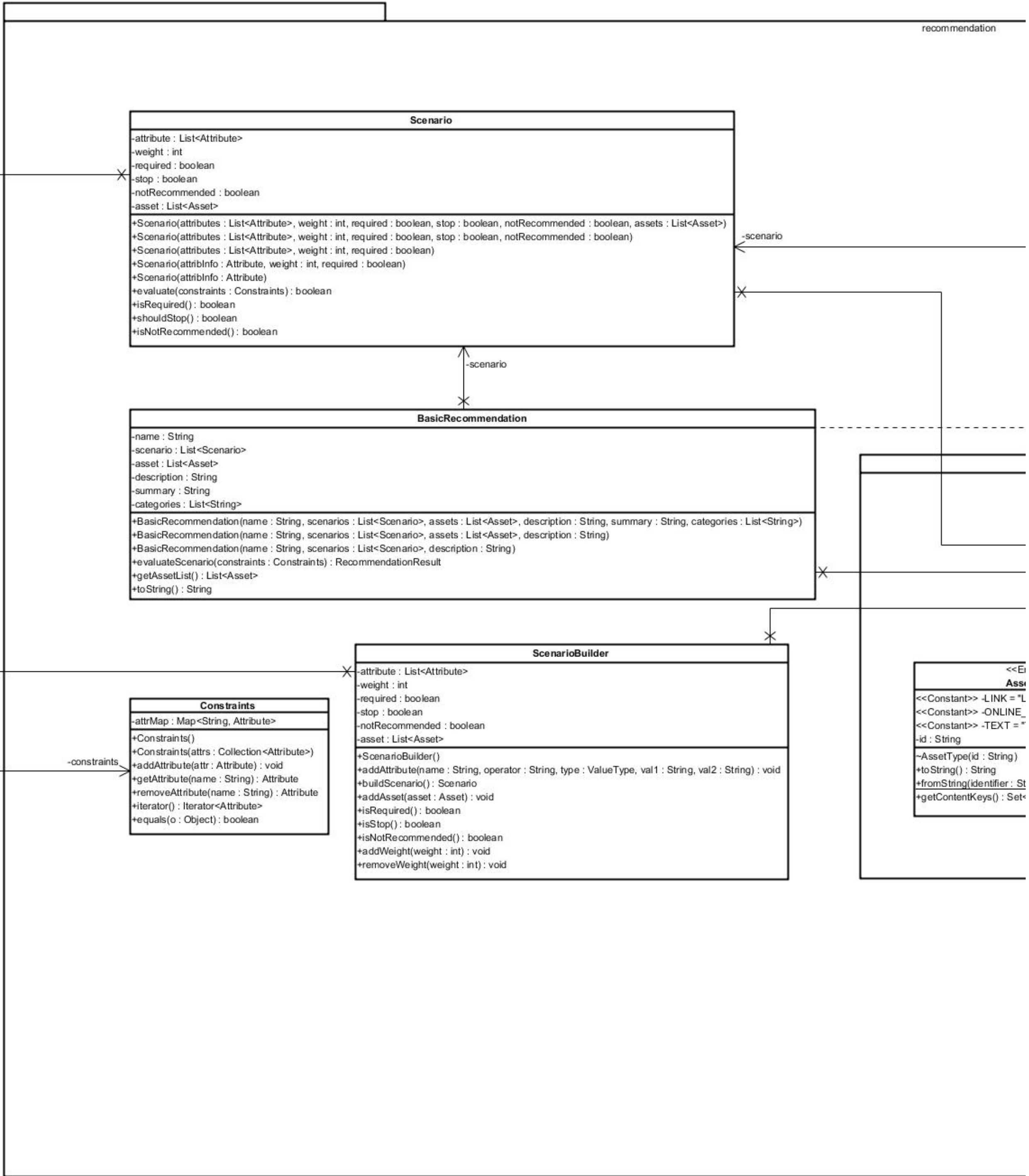
-recs : Collection<Recommendation>
-attrRef : AttributeReference

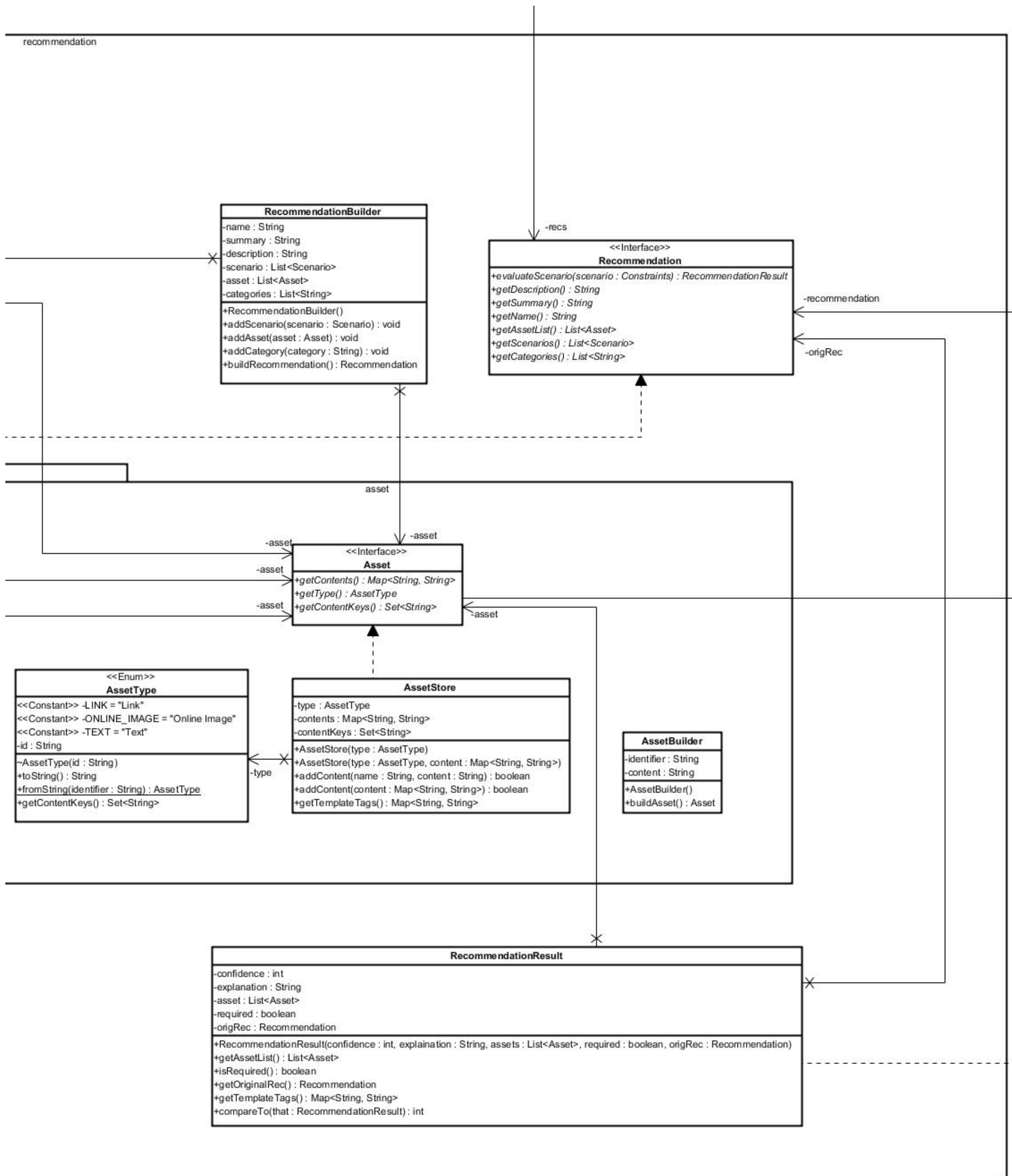
+TestDatabase(recs : Collection<Recommendation>, attrRef : AttributeReference)
+TestDatabase()
+use(dbName : String) : void
+close() : void
+getAllRecommendations() : Collection<Recommendation>
+getAttributeReference() : AttributeReference
+store(o : Object) : boolean
+delete(o : Object) : boolean
+getRecommendation(name : String) : Recommendation

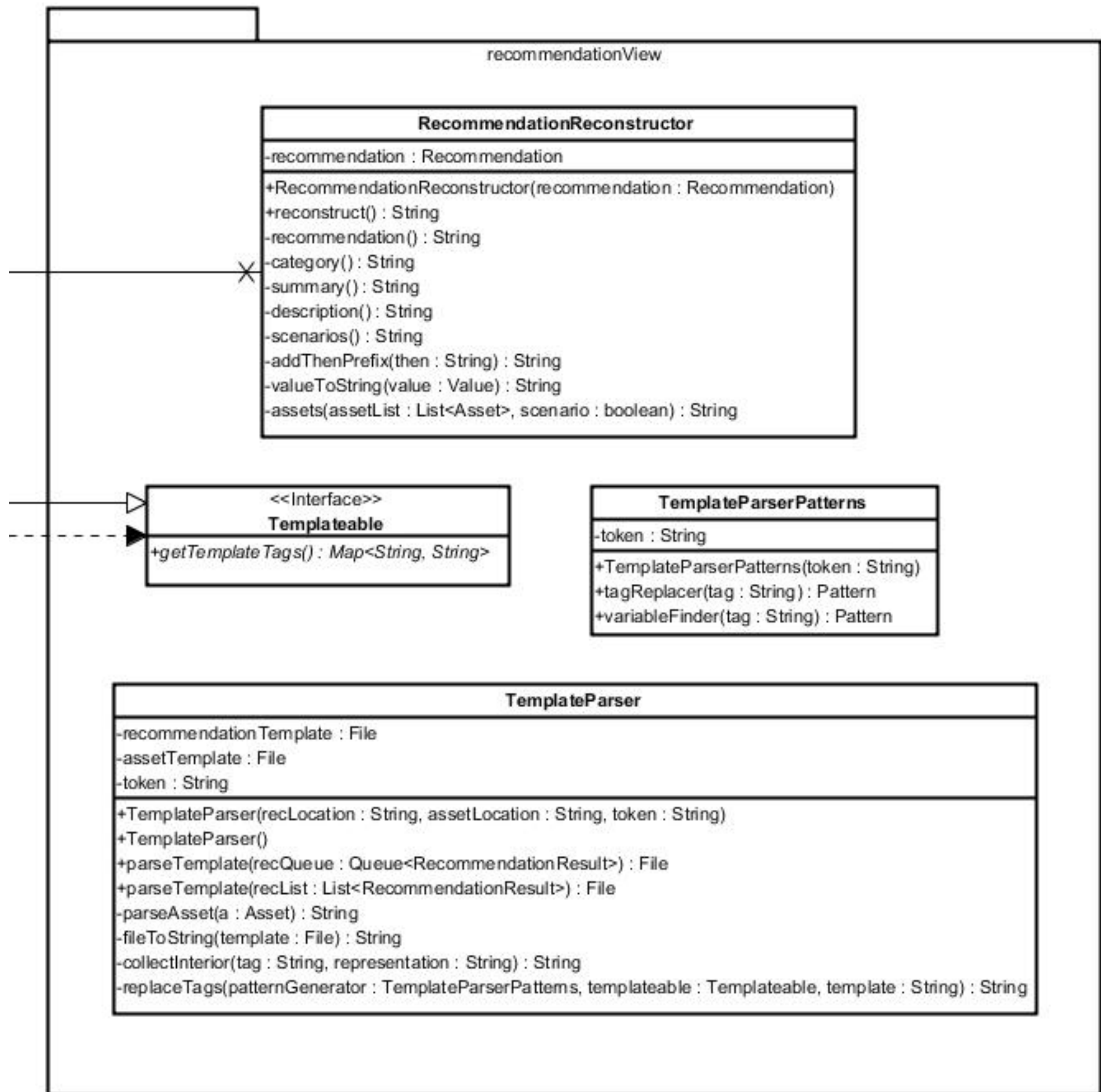
DatabaseItemNotFoundException

+DatabaseItemNotFoundException(msg : String)


dialog
ConsoleController
+runDialog() : void
+runSubDialog(subDialog : SubDialog) : void
-displayQuestion(question : Question) : void
-readAnswerFromUser(question : Question) : void
UniqueQuestionIdException
+UniqueQuestionIdException(msg : String)
Answer
-answerNum : int
-answerText : String
-attrValue : Value
-parentQuestion : Question
-isSelected : boolean
-attr : Attribute = null
+Answer(answerNum : int, text : String, attrValue : Value)
+Answer(answerNum : int, text : String)
+select() : void
+deselect() : void
+toString() : String
+value() : boolean
Dialog
-subDialogList : List<SubDialog> = null
-constraints : Constraints = null
-database : Database = null
-attrRef : AttributeReference = null
-currentQuestion : Question = null
-Dialog(database : Database)
+startDialog(database : Database) : void
+addSubDialog(subDialog : SubDialog) : void
+findSubDialogByTitle(title : String) : SubDialog
+addAttribute(attr : Attribute) : void
+removeAttribute(attr : Attribute) : void
+getAttribute(name : String) : Attribute
+getNextQuestion() : Question
+close() : void
+getAttributeReference() : AttributeReference
+getResultingRecommendations() : PriorityQueue<RecommendationResult>
-currentQuestion
-parentQuestion
-answerList
-subDialogList
Question
-id : String
-questionStatement : String
-answerList : List<Answer>
-parentSubDialog : SubDialog
-trigger : BooleanValue
-attrDesc : AttributeDescription
+Question(text : String, attrDesc : AttributeDescription)
+Question(text : String)
+setID(id : String) : void
+addAnswer(answer : Answer) : void
~getId() : String
+getAttributeDescription() : AttributeDescription
+equals(input : Object) : boolean
+toString() : String
-currentQuestion
-parentSubDialog
-questionList
SubDialog
-title : String
-questionList : List<Question>
-trigger : BooleanValue
+SubDialog(title : String)
+addQuestion(question : Question) : void
+getQuestionById(id : String) : Question
+equals(input : Object) : boolean
+toString() : String
-subDialog


recommendation
Scenario
-attribute : List<Attribute>
-weight : int
-required : boolean
-stop : boolean
-notRecommended : boolean
-asset : List<Asset>
+Scenario(attributes : List<Attribute>, weight : int, required : boolean, stop : boolean, notRecommended : boolean, assets : List<Asset>)
+Scenario(attributes : List<Attribute>, weight : int, required : boolean, stop : boolean, notRecommended : boolean)
+Scenario(attributes : List<Attribute>, weight : int, required : boolean)
+Scenario(attribInfo : Attribute, weight : int, required : boolean)
+Scenario(attribInfo : Attribute)
+evaluate(constraints : Constraints) : boolean
+isRequired() : boolean
+shouldStop() : boolean
+isNotRecommended() : boolean
-scenario
-scenario
BasicRecommendation
-name : String
-scenario : List<Scenario>
-asset : List<Asset>
-description : String
-summary : String
-categories : List<String>
+BasicRecommendation(name : String, scenarios : List<Scenario>, assets : List<Asset>, description : String, summary : String, categories : List<String>)
+BasicRecommendation(name : String, scenarios : List<Scenario>, assets : List<Asset>, description : String)
+BasicRecommendation(name : String, scenarios : List<Scenario>, description : String)
+evaluateScenario(constraints : Constraints) : RecommendationResult
+getAssetList() : List<Asset>
+toString() : String
ScenarioBuilder
-attribute : List<Attribute>
-weight : int
-required : boolean
-stop : boolean
-notRecommended : boolean
-asset : List<Asset>
+ScenarioBuilder()
+addAttribute(name : String, operator : String, type : ValueType, val1 : String, val2 : String) : void
+buildScenario() : Scenario
+addAsset(asset : Asset) : void
+isRequired() : boolean
+isStop() : boolean
+isNotRecommended() : boolean
+addWeight(weight : int) : void
+removeWeight(weight : int) : void
Constraints
-attrMap : Map<String, Attribute>
+Constraints()
+Constraints(attrs : Collection<Attribute>)
+addAttribute(attr : Attribute) : void
+getAttribute(name : String) : Attribute
+removeAttribute(name : String) : Attribute
+iterator() : Iterator<Attribute>
+equals(o : Object) : boolean
-constraints
<<Constant>> -LINK =
<<Constant>> -ONLINE_
<<Constant>> -TEXT =
-id : String
~AssetType(id : String)
+toString() : String
+getContentKeys() : Set


recommendation
RecommendationBuilder
-name : String
-summary : String
-description : String
-scenario : List<Scenario>
-asset : List<Asset>
-categories : List<String>
+RecommendationBuilder()
+addScenario(scenario : Scenario) : void
+addAsset(asset : Asset) : void
+addCategory(category : String) : void
+buildRecommendation() : Recommendation
-recs
<<Interface>>
Recommendation
+evaluateScenario(scenario : Constraints) : RecommendationResult
+getDescription() : String
+getSummary() : String
+getName() : String
+getAssetList() : List<Asset>
+getScenarios() : List<Scenario>
+getCategories() : List<String>
-recommendation
-origRec
asset
-asset
<<Interface>>
Asset
+getContents() : Map<String, String>
+getType() : AssetType
+getContentKeys() : Set<String>
<<Enum>>
AssetType
<<Constant>> -LINK = "Link"
<<Constant>> -ONLINE_IMAGE = "Online Image"
<<Constant>> -TEXT = "Text"
-id : String
~AssetType(id : String)
+toString() : String
+fromString(identifier : String) : AssetType
+getContentKeys() : Set<String>
-type
AssetStore
-type : AssetType
-contents : Map<String, String>
-contentKeys : Set<String>
+AssetStore(type : AssetType)
+AssetStore(type : AssetType, content : Map<String, String>)
+addContent(name : String, content : String) : boolean
+addContent(content : Map<String, String>) : boolean
+getTemplateTags() : Map<String, String>
AssetBuilder
-identifier : String
-content : String
+AssetBuilder()
+buildAsset() : Asset
RecommendationResult
-confidence : int
-explanation : String
-asset : List<Asset>
-required : boolean
-origRec : Recommendation
+RecommendationResult(confidence : int, explaination : String, assets : List<Asset>, required : boolean, origRec : Recommendation)
+getAssetList() : List<Asset>
+isRequired() : boolean
+getOriginalRec() : Recommendation
+getTemplateTags() : Map<String, String>
+compareTo(that : RecommendationResult) : int


recommendationView
RecommendationReconstructor
-recommendation : Recommendation
+RecommendationReconstructor(recommendation : Recommendation)
+reconstruct() : String
-recommendation() : String
-category() : String
-summary() : String
-description() : String
-scenarios() : String
-addThenPrefix(then : String) : String
-valueToString(value : Value) : String
-assets(assetList : List<Asset>, scenario : boolean) : String
<<Interface>>
Templateable
+getTemplateTags() : Map<String, String>
TemplateParserPatterns
-token : String
+TemplateParserPatterns(token : String)
+tagReplacer(tag : String) : Pattern
+variableFinder(tag : String) : Pattern
TemplateParser
-recommendationTemplate : File
-assetTemplate : File
-token : String
+TemplateParser(recLocation : String, assetLocation : String, token : String)
+TemplateParser()
+parseTemplate(recQueue : Queue<RecommendationResult>) : File
+parseTemplate(recList : List<RecommendationResult>) : File
-parseAsset(a : Asset) : String
-fileToString(template : File) : String
-collectInterior(tag : String, representation : String) : String
-replaceTags(patternGenerator : TemplateParserPatterns, templateable : Templateable, template : String) : String

Appendix C Software Engineering Survey

The following survey was sent to Computer Science, Robotics, and Interactive Media and Game Development majors at Worcester Polytechnic Institute. Responses to this survey were used to help construct Recommendations for the Software Process Configurator.

Software Process Configurator MQP

This survey is intended for those who have taken CS 3733, Software Engineering. It will ask about the effectiveness of software development practices used during that class.

All questions are optional, but if you feel very strongly that a particular strategy, tool, or practice was extremely helpful or harmful to your project, please leave additional comments and tell us why.

Thank you for your time.

Please provide a brief description of your project, and when you took the class:

Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Software Process Configurator MQP

Instructions (Please Read!)

We wish to know how effective each development practice was. By effective, we mean that using the practice made it easier to complete your project, or made the final result better in some way.

For each development practice, please rate its effectiveness for your project. Skip any question that doesn't apply, or that you have no opinion on.

How effective was dividing the project into weekly iterations?

	1	2	3	4	5	
Not Very Effective	○	○	○	○	○	Very Effective

If it was not effective, how would you change it?

Did you assign roles to any team members, and if so, how useful was it?

Overall, how effective was the "Planning game?"
Breaking up work into user stories with points, assigning them, budgeting your work, etc.

	1	2	3	4	5	
Not Very Effective	○	○	○	○	○	Very Effective

What were the most and least useful aspects of the planning game?

Please add any additional comments you may have.
Helpful comments include why you considered a particular practice effective or what you would have done differently.

« Back Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Software Process Configurator MQP

Instructions (Page 2 of 3)

For each type of development practice, please specify what you used during your project. Then rate and comment as you did in the previous section.

What languages and development environments did you use to make your project?

For example, using Java and Eclipse

How effective was this choice of language?

	1	2	3	4	5	
Not Very Effective	○	○	○	○	○	Very Effective

What would you change about your choice if you had to do the project again?

Assume that it is the exact same project.

What did you use for version control, and why did you choose it?

E.g.: SVN, Git

How effective was using this choice of version control?

	1	2	3	4	5	
Not Very Effective	○	○	○	○	○	Very Effective

How was the work divided up among your team members?
In terms of assigning user stories to particular members, or groups, and the general work-completion strategy.

How effective was this division of labor?

	1	2	3	4	5	
Not Very Effective	○	○	○	○	○	Very Effective

Why did you divide up the work this way? If it was not effective, how would you change it?

How often did the entire group, or any subgroups, meet?

If you used Continuous Integration, how effective did you find this to be?
E.g. Jenkins, Hudson.

	1	2	3	4	5	
Not Very Effective	◯	◯	◯	◯	◯	Very Effective

If you used a form of bug tracking, how effective was it in terms of reducing the number of bugs and streamlining the project?

	1	2	3	4	5	
Not Very Effective	◯	◯	◯	◯	◯	Very Effective

Do you have any comments about continuous integration or bug tracking?

Please add any additional comments you may have.
Helpful comments include why you chose what you did and what you would choose given another chance.

« Back   Continue »

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Software Process Configurator MQP

Instructions (Last page!)

If you used any additional tools or strategies during your project, please include them below.

For any additional tools or strategies, please list them below and comment on how effective they were.

Thank you for participating in our survey!

If you wish to provide more information, or you are curious and want to know more, or if you have any questions, please leave your email address below and we will contact you.

Email Address (Optional)

Your address will remain confidential and will never be associated with your other responses.

« Back Submit

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

Appendix D Grammar for the Domain Specific Language

```
Recommendation -> 'Recommendation:' Name Statement+

Statement -> Category | Summary | Description | Scenario | Asset

Category -> 'Category:' Name

Summary -> 'Summary:' Text

Description -> 'Description:' Text

Asset -> AssetType 'Asset:' KeyValue+

AssetType -> 'Link' | 'Text' | 'Online' 'Image'

KeyValue -> Name ':' Text      //See bottom for note

Scenario -> 'Scenario:' 'Given' ExpressionList 'Then' ActionList

ExpressionList -> Expression ('And' Expression)*

Expression -> Name OperatorList

OperatorList -> Operator ('Or' Operator)*

Operator -> SingleOp Value | DoubleOp Value 'And' Value

SingleOp -> Equal | NotEqual | GreaterThan | LessThan |
            GreaterThanEqual | LessThanEqual

DoubleOp -> Between

ActionList -> Action ('And' Action)*

Action -> AddRemoveConfidence | 'Stop' | 'Make' 'Required' | 'Do'
          'Not' 'Recommend' | AddAsset

AddRemoveConfidence -> ('Add' | 'Remove') Integer 'Confidence'

AddAsset -> 'Add' '{' Asset '}'

--------------------------------

Equal -> 'is' | 'is'? ('=' | 'equal') | 'equals'

NotEquals -> 'is'? ('!=' | 'not' 'equal') | 'is' 'not'

GreaterThan -> 'is'? ('>' | 'greater' 'than')

LessThan -> 'is'? ('<' | 'less' 'than')
```

```
GreaterThanEqual -> 'is'? ('>=' | 'greater' 'than' 'or' 'equal' 'to')

LessThanEqual -> 'is'? ('<=' | 'less' 'than' 'or' 'equal' 'to')

Between -> 'is'? 'between'

Value -> Name | Integer | Boolean | String

Boolean -> 'true' | 'false'

String -> '"' Text '"'

Integer -> Digit+

Name -> Word+

Word -> Letter | Digit | '_' | '-' | '''

Digit -> '0'..'9'

Letter -> 'a'..'z' | 'A'..'Z'

Text -> .*

/*
NOTE: For assets the valid Names in the KeyValue depends on the type.
      For Link names can be 'url' and 'text'
      For Text the only name available is 'text'
      For Online Image names can be 'src', 'alt', and 'caption'

I could have written this out in grammar form but it would have been
very verbose and possibly confusing.  Here is an example of a link
asset in case you are still confused:

Link Asset:
     url: www.google.com
     text: Google Web Search
*/
```

Appendix E Recommendation Examples

```
Recommendation: Git
Category: SoftEngProject
Summary: Version Control System - Git
Description:
Git is a source code management and version control tool.  It is free, fast,
and used for many applications.

Scenario:
    Given mustUseGit is true
    Then Make Required

Scenario:
    Given hasGitUser is true
    Then Add 25 confidence

Link Asset:
    URL: http://en.wikipedia.org/wiki/Git_(software)
    Text: Git - Wikipedia

Link Asset:
    URL: http://git-scm.com/
    Text: Git main webpage

Link Asset:
    URL: http://www.syntevo.com/smartgit/index.html
    Text: SmartGit - Graphical client for Git
```

```
Recommendation: Project Manager
Category: SoftEngProject
Summary: Team member with a dedicated Project Manager role
Description:
A Project Manager is a team member who is in charge of the team itself.  He or
 she should organize meetings, provide agendas, schedule work, and help point
the team in the right direction.

Scenario:
    Given teamSize is between 5 and 20
    And deadline = true
    And projectTimeWeeks > 3
    Then Add 20 confidence

Scenario:
    Given teamSize >= 10
    Then Add 30 confidence

Link Asset:
    URL: http://en.wikipedia.org/wiki/Project_manager
    Text: Project Manager - Wikipedia
```

```
Recommendation: Short Iterations
Category: SoftEngProject
Summary: Iteration-based project development
Description:
By splitting up your project into set-length "iterations," project development
 can be more organized.  It also encourages productivity.
For a project of this nature, we recommend using 1-2 week iterations,
depending on your time commitments.

Scenario:
    Given teamSize is between 5 and 20
    And deadline = true
    And projectTimeWeeks > 3
    Then Add 50 confidence

Scenario:
    Given teamSize is between 5 and 20
    And deadline = false
    Then Add 30 confidence

Link Asset:
    URL: http://en.wikipedia.org/wiki/Iterative_and_incremental_development
    Text: Iterative and Incremental Development - Wikipedia
```

```
Recommendation: Meet Often As An Entire Group

Summary: As often as possible meet as an entire group. This helps everyone know what
is going on and help each other out.

Description:
    It may seem like meeting as an entire group many times a week would be a waste of
     time, but it can actually be very helpful. Group meetings can be used for
    planning the week, but they can also be working meetings where everyone can work
    in the same room and get help and clarification when needed. This eliminates
    wasted time when people get stuck or are not sure what they need to do.

Link Asset:
    url: http://web.cs.wpi.edu/~gpollice/testimonials.html
    text: Testimonials from Past Students

Scenario:
    Given team_experience <= 5 And
        meeting_days_per_week > 3
    Then Add 100 confidence

Scenario:
    Given team_experience > 8
    Then Remove 50 Confidence

Scenario:
    Given meeting_days_per_week >= 5
    Then Add { Link Asset:
        url: http://en.wikipedia.org/wiki/Stand-up_meeting
        text: Try a daily standup meeting }
```