March 2012

# Software Process Configurator

Jake William Todaro
*Worcester Polytechnic Institute*

Pragathi Balasubramanian
*Worcester Polytechnic Institute*

Project Number: GFP1104

Software Process Configurator

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:


____________________________

Pragathi Balasubramanian

____________________________

Jake Todaro


Date: March 10, 2012

Approved:

____________________________

Professor Gary F. Pollice, Major Advisor

## Contents

Abstract ........ 3

Acknowledgements ........ 4

Introduction ........ 5

Background ........ 8

Software Development Processes ........ 8

What is a software development process? ........ 9

Choosing between Processes ........ 12

Existing Processes ........ 14

Decision Support System (DSS) ........ 21

The History and Definition of DSS ........ 21

DSS Characteristics and Components ........ 22

Types of DSS - Expert Systems ........ 22

Background Summary ........ 24

Methodology ........ 26

Problem Statement ........ 26

Goals ........ 26

User Scenarios ........ 27

Project Set Up ........ 28

Design 1 - Visualizing Dialog as a Directed Graph ........ 28

Design ........ 29

Implementation ........ 29

Design 2 - Visualizing Dialog as a Linear list ........ 30

Design ........ 30

Design 3 - Final Architecture ........ 31

Dialog ........ 31

Dialog Writing ........ 33

Graphical User Interface (GUI) ........ 35

Results ........ 36

Future work ........ 38

Dealing with Dialog Errors ........ 38

Future Forms of Dialog Inputs ........ 38

Asset Database Editor .......... 38
Results Templates .......... 39
Asset Categorization .......... 39
Complex Recommendations .......... 40
Testing for Proper Results .......... 40
Works Cited .......... 41
Appendix A – Twelve Principles of Agile .......... 47
Appendix B – How the code works – Developers Guide .......... 48
Overview .......... 48
The Asset Database .......... 48
The Dialog .......... 48
Inputs .......... 50
The Trigger System .......... 50
Appendix C – XML Dialog Tutorial .......... 51
Index file .......... 51
SubDialog Basics .......... 51
Recommendations .......... 52
Triggers .......... 52
Appendix D – “Where should I eat?” Test Dialog .......... 55

# Abstract

This paper describes the Software Process Configuration (SPC) and the work surrounding its creation. The SPC is a program that uses domain expert knowledge on software processes to provide tools and information to non-experts who would like help managing a software project. The SPC uses dialog based guidance to gather user input on a software development scenario. The tools and information provided to the user at the end is gathered using a heuristic embedded in the dialog. The SPC is still a proof of concept implementation and can be customized to any decision making scenario.

## Acknowledgements

We would like to thank our advisor Professor Gary Pollice for his invaluable guidance and expertise in the field of software development.

## Introduction

One can describe a software development process differently in the different contexts they are used. It may involve a methodology, the stakeholders, and the technology necessary to execute it, or it may only involve a document outlining the procedure, the knowledge necessary for the process and its results. Generally, a software development process catering to both the business end and the development end includes a lifecycle process for both business and technological issues, concepts, models, rules, deliverables, guidelines for project management, and identification of organizational roles (Hull, Taylor, Hanah & Millar, 2002).

Numerous software development processes exist that cater to different software development environments. Each development process has a different structure and discipline attributed to it. In many cases, the exact implementation details of the process are vague and left up to the project leaders or software developers to decide what is appropriate. The varying applications of software makes determining the most appropriate development process for a specific software project difficult. Process engineers and project managers in charge of developing software who face this dilemma cannot decide which development methodology is most suited for their project (G. Pollice, personal communication, 2011).

In order to completely understand the different development processes, and have the knowledge to pick the appropriate method, one would need software process engineering expertise. Employing decision-making software, or a decision support system (DSS), enables information transfer from a process expert to those without expert knowledge. These systems aid users in making decisions and solving problems by utilizing computer communications, data, documents, knowledge, and models (Power & Sharda, 2009). Such decision software can provide key process guidance when supplemented with information by a development expert. Software development specialists would have the knowledge base and access to resources necessary to recommend a development process for different project scenarios.

Process experts in the software development domain could break down the problem of finding the appropriate development methodology into a set of smaller decisions or questions. In this scenario, process engineers with such a tool can eliminate the time needed to research and understand the numerous processes available, and focus on the smaller questions devised by experts. These questions would then lead to a strategic decision on what development methodology and resources would be necessary for a particular scenario. Due to the phased nature of most development processes, experts can divide the process into disciplines separate disciplines like requirements engineering, design and implementation, testing, and release. Through this breakdown, a dialog, a set of relevant information, and helpful tools can be compiled by experts to create a DSS capable of guiding the choice of a software development process.

Our goal for this project is to create a system that allows software development experts to provide necessary tools and information to help those less proficient in choosing and implementing a software development process. We call this system the Software Process Configurator (SPC). Our objectives for the SPC include creating:

- a framework for a “smart” dialog
- a database to hold information and tools relevant to software development processes
- an input method for creating a dialog
- a user interface for the SPC

To achieve our objective we researched software development processes, DSS’s, and gathered information from Professor Gary Pollice, who is an expert in software development. Using the information collected from research, we developed a dialog based guidance software application that provides advice on selecting software processes, practices and tools appropriate for a variety of software projects. Using the SPC we were able to input dialog provided by Professor Pollice to construct a decision making model that chooses between several software development processes. The SPC is a proof of concept, showing that decision making software can be applied to assist during complex decisions in the context of software development.

# Background

This background focuses on select software development processes, and decision support systems. This research helped us formulate the requirements necessary for developing decision making software capable of choosing between software development processes.

## Software Development Processes

The history of software development began when software professionals defined software engineering. The world saw the need for software in modern society as it helped provide more efficient solutions to problems present in anything from business to medicine (Kruchten, 1998). This need caused professionals to coin the term software engineering when the first international conference on this topic was held in October 1968. The discipline has come a long way since then, and is now defined by the IEEE Computer Society as:

> “(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)." (IEEE, 2004, p.1)

Along with the definition of software engineering came the necessity for structure in software development. This resulted in the software development life cycle (SDLC) that provides a set of stages for software development. According to the IEEE, in software engineering, five of eleven main knowledge areas include the different phases of the SDLC. The SDLC typically consists of the following phases: requirements engineering, design and implementation, testing, release, and maintenance (Petersen, Wohlin, & Baca, 2009). Although every process performs each of those phases differently, different processes emphasize different activities in the cycle. Some provide detailed instructions for every stage in the SDLC, while others focus on key areas deemed the most important by the process creators.

This section describes the software development process, and its role to set a standard and a basis for comparison between different processes. We describe the SDLC, and how it pertains to existing software processes. We also address the dilemma of choosing between development processes. This emphasizes a project manager's need for help from domain experts. Finally, we address the existing software development process and how they are enforced.

## What is a software development process?

When defining a software development process, the fundamental concepts behind a process must first be defined. One definition states that process consists of who does what, when and how (Fayad, 1997). Meanwhile, the Merriam-Webster Collegiate Dictionary describes it as "a series of actions or operations conducing to an end". Another Webster definition states that a process usually involves a collection of sets or operations (Merriam-Webster, 2012). In the article *Process Diversity in Software Development,* Lindvall and Rus (2000) argue that the definition of a process depends on its end goal. When the different definitions are consolidated, a process can be loosely defined as a set of predetermined steps for all those involved, towards a particular goal. In the case of using processes in industry, the end goal usually determines the steps taken in the process depending on scope and organizational level. Moreover, a process should provide guidance for dividing tasks, coordinating tasks, and communication among all those involved (Lindvall & Rus, 2000).

When applied to software development, a process must provide an operating procedure for everyone involved in a software project. This structured approach is designed to promote a common vision and increase productivity by guiding a project through the stages of the software life cycle. One can describe software development as consisting of methods, the activities performed by software engineers, such as use case diagrams, state transition diagrams, etc. When we consider these methods together we refer to them as a methodology (Hull, Taylor, Hanna, & Miller, 2002). So, a software development process is a guide for software projects and all of the roles involved in the project through the software development life cycle.

The SDLC provides a general structure for any software development, and has especially proven useful in the shaping of the different development methods existent today. We now look at the common phases of all SDLCs.

## *Requirements Engineering*

This phase involves defining what a software or system must do, its characteristics and the constraints under which it should operate (Kotonya & Sommerville, 1998). The development team first identifies the users, the customers and other stakeholders of the software who can provide such information. Once they are familiar with the stakeholders, the team identifies their needs and incorporate them into list of requirements of the system being built. The team might utilize use cases, and/or user stories to help them identify the requirements. On the other hand, the end user might directly hand the requirements to the development team (Petersen et al., 2009).

We use different types of requirements for different purposes. The following list identifies a few of the common types:

- General requirements are very broad and explain what the software or system must be capable of.
- Functional requirements describe a process the software must perform or a particular piece information it must have (Dennis et al., 2010).
- Implementation requirements specify how the system must be implemented.
- Performance requirements determine the minimum acceptable performance for a system, such as response time, capacity, and reliability.

### *Design and Implementation*

The design phase entails planning the architecture of the software, and how the architecture will be implemented. This stage includes writing use cases, designing UML diagrams, and configuring the inputs and outputs of each software module. Once the software blueprint is finalized, the implementation of the software begins. During implementation, the team commences with the implementation. The team performs basic unit testing before passing the software on to testing. In addition, the team will also verify that the developed software adheres to all requirements and is capable of satisfying the user and customer's needs (Petersen et al., 2010).

### *Testing*

Testing involves ensuring that the software meets all requirements, and runs as expected. In some software development processes, testing is encouraged as soon as development begins. There are many types of testing that are used to check different aspects of a system or software. The tests employed for any software project depend on the type of software being developed. The following list illustrates different types of tests:

- Module(Unit) Testing tests the different modules of the software. A module is any set of program statements that can be compiled independent from other modules (Li, 1990).
- Integration Testing involves merging and testing program modules to check that they work as a whole (Li, 1990).
- Software Acceptance Testing checks if users are able to accept the software and if a formal certification can be issued (Li, 1990).
- Structural and Functional Testing checks that architecture and functionality of the software works as expected. These may include checking for efficiency under different workloads, reliability, and security testing (Li, 1990).

- Installation Testing ensures that the software can be installed and run with no errors (Li, 1990).

### *Release*

In the release stage of the SDLC, requirements and functional testing occurs. In this phase, the team checks that the users and customers are satisfied with the software and approve that the final outcome has met their requirements (Li, 1990).

### *Maintenance*

The final stage of the SDLC assures that the software has a support system where errors can be reported and fixed. The team must establish a structure as to how and when the software updates occur. Furthermore, they have to arrange a form of communication between the users, customers and the team itself for any future issues with the software (Li, 1990).

The SDLC skeleton applies to many existing development processes. Although its exact application varies between each, the purpose of every stage is usually fulfilled.

## Choosing between Processes

There is such a large variety of software process today that it could be very time consuming and difficult to pick out the best process for a particular project (Lindvall & Rus, 2000). Experts argue that there are many different factors involved in picking out the right process. Lindvall and Rus (2000) state that project goal, and available resources determine the type of process needed. On the other hand, Cockburn (2000) believes that a process is chosen by three main factors, project size, criticality of the system, and the project’s priorities. Alexander and Davis (1991) developed criteria to select the appropriate type of process needed for a particular project. These criteria involve a more mathematical approach with criteria and importance represented in matrices. Addressing and comparing these ideas provides a larger to context as to why choosing the correct process is difficult.

In the article *Process Diversity in Software Development*, the authors mention that depending on the organizational level, the project's goals and available resources affect the process chosen. The resources being the size of the company, the number, knowledge, and experience of the people involved, and available hardware. However, they state that finding the best process for a particular scenario "is the most difficult question". They present a set of 16 articles that provide case studies on different companies, and the software development process they employ for their development projects. These articles showcase the different experiences of those in the industry and what they have learnt through implementing software processes. Although reading these articles might assist a development team find the perfect process, it would be time consuming and may not provide enough information. Ultimately, Lindvall and Rus (2000) agree that the team in charge of the development must be aware of all that their project entails, and the variety of processes available, in order to choose the best software development process (Lindvall & Rus, 2000).

Cockburn (2000) performs a similar study where he consolidates the selection of the right software process into a methodology grid. He devises this grid through four main principles and two important factors. The principles define the importance of the criticality of the system and the size of project. The other two factors are the project priorities and the methodology designer. Cockburn mentions from experience that finding a project's priorities is no easy task. Furthermore, he assumes that there is a designer in the team assigned to develop a methodology based on his/her experience. He creates the methodology grid to help the designer determine the size and density of the process methodology. However, in the case of a general project manager, there may not always be a methodology designer to find the best process. According to this article, there would still be a need for an expert in the matter of software development to make the final choice (Cockburn, 2000).

The last study we reviewed on software process selection provided a mathematical approach using 20 criteria. These project criteria are determined by the personnel involved in the project, the problem, the product, the resources, and the organizational factors. These criteria culminate to form a matrix of values depending on how well a process suits a project. However, in this study by Alexander and Davis, the processes being chosen are on a more general organizational level. In fact, there are only three processes specific to development that can be chosen through their layout. Their conclusion points out that further studies still need to be done to provide valid correlations between a project's characteristics and the chosen process (Alexander & Davis, 1991).

These studies show that finding the best fit process for a software project is not a simple procedure. One must take a variety of factors taken into consideration before reaching a final decision. Through these articles, we can conclude that experience and expertise in the field software processes are needed to find the most suitable process for a given software project.

## Existing Processes

There are many processes in the software development ecosystem. Out of those, there are three main categories: Plan-driven, Iterative, and Agile. For each of those categories we have picked out one main development process. The waterfall development process that follows the SDLC stages sequentially best illustrates the concepts of a plan-driven process. For iterative development, we chose to describe the Rational Unified Process (RUP). Finally, for Agile processes, we chose to describe SCRUM and eXtreme programming.

### *Waterfall Development Process*

The waterfall development method is the simplest of all software development processes. It follows the SDLC structure exactly and in a sequential manner as seen in Figure 1-1. As in the phases of the SDLC, all the requirements are specified before design begins and design completes before implementation. This process only allows limited changes to requirements when proceeding through other phases of SDLC.

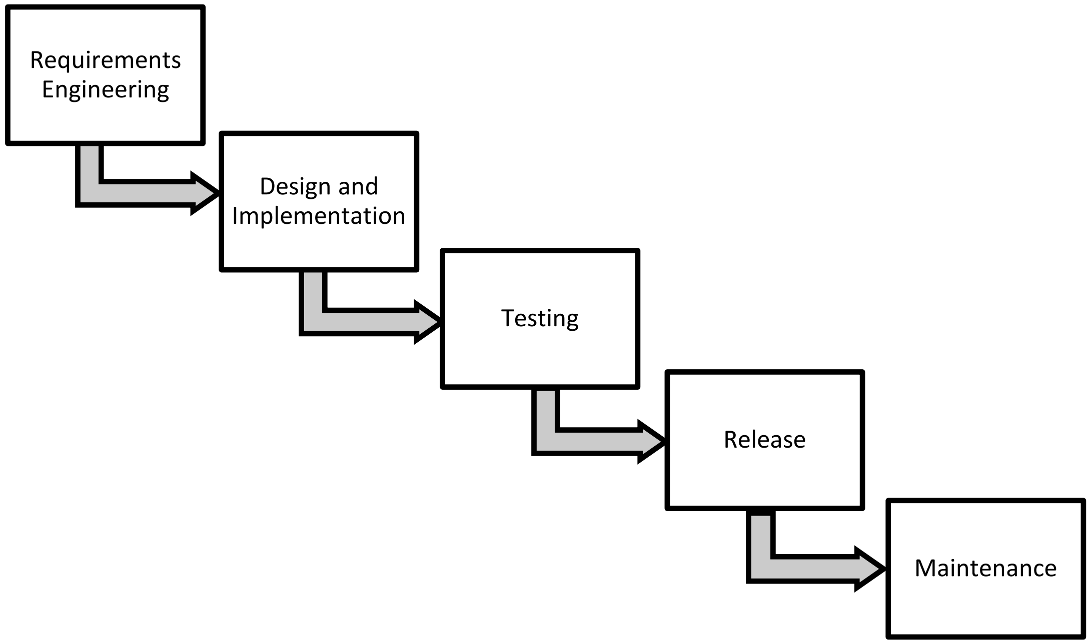


**Figure 1-1 Waterfall SDLC**

This development strategy is generally avoided since requirements may get overlooked during long projects. The sequential phases also decrease adaptability of project, especially long term ones (Northrop, 2004, p. 40). This process has two important variations, *Parallel Development* and *V-model*. The parallel development variation was designed to reduce the time consumed for traditional waterfall development. The software being developed is broken into smaller projects which are all designed and implemented in parallel. At the end, all the subprojects are integrated with each other, tested, and then handed off to the customer. The *V-model* is a variation of the traditional waterfall process that implements a specific testing process. As the software development goes through its SDLC phases, tests will be written simultaneously to guarantee better quality software. However, since the V-model is sequential, it incurs all the major disadvantages of the traditional waterfall model.

### *Rational Unified Process (RUP)*

As defined by Kruchten (1998), RUP is a software engineering process that "provides a disciplined approach to assigning tasks and responsibilities within a development organization". Its goal is to be able to use a predictable schedule and a reasonable budget as means to create high quality software that satisfies the users (Kruchten, 1998, p.17). RUP has a process framework that can be adapted to any software project. Furthermore, RUP is heavily use-case driven; use cases provide the link between system requirements and the designing and testing portions. This process focuses on a set of software development best practices (Krutchen, 1998):

1. Develop software iteratively
2. Manage requirements
3. Use component-based architectures
4. Visually model software
5. Verify software quality
6. Control changes to software

There are three main elements in RUP:

- *Roles* - behavior and responsibilities of an individual or a group
- *Activities* - the behavior of the worker
- *Artifacts* - what the worker creates modifies or controls.

The *workflow*[1] is what brings the three elements together. A workflow is a sequence of activities that produces a particular result. A complete RUP lifecycle is composed of workflows and iterations. The

[1] The term *workflow* has been changed to *discipline* in the RUP process.

main workflows in RUP are organized under Core Workflows, Iteration Workflows and Workflow details.

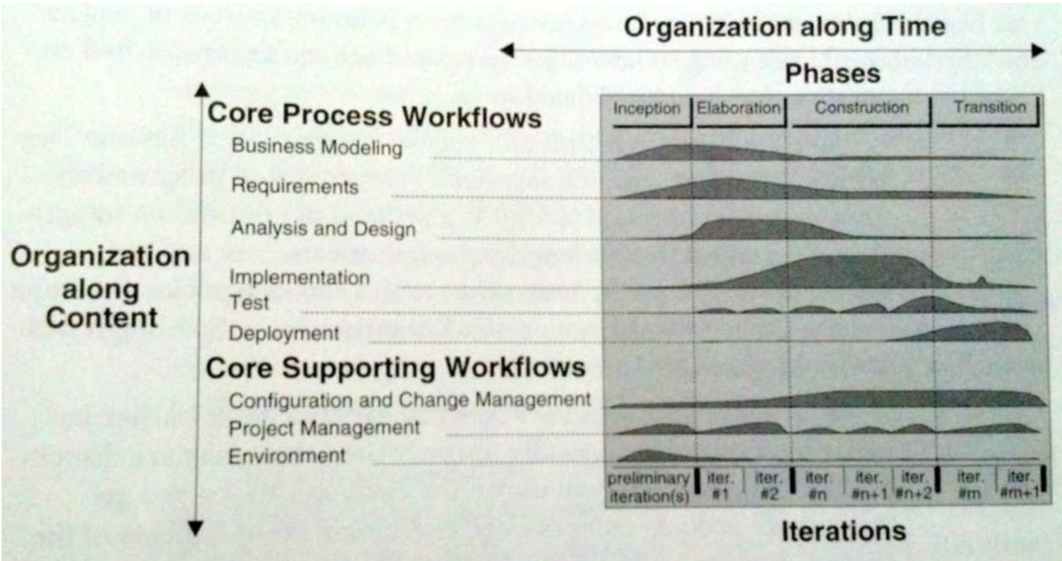


**Figure 1-2 RUP Workflow Chart (Krutchen, 1998)**

The core workflows are illustrated in Figure 1-2. As seen in the diagram, the core process workflows or the engineering workflows and the supporting workflows compose the nine RUP workflows. The words labeling vertical columns on the top represent the phases in a RUP process. Meanwhile, since RUP is an iterative process, there are a number of iterations each phase goes through. Each of these iterations consists of an iteration workflow. The iteration workflow describes each iteration of a particular phase. These can be seen in Figure 1-2 as the smaller "iter" portions labeled in the bottom. Finally, the workflow details elaborate on closely related activities, the input and output information flow, and how activities interact through artifacts (Kruchten, 1998).

### *Agile Development*

Agile development is a group of programming centric processes that focuses on face-to-face communication (Dennis et al., 2010, p. 50). This form of development came about as a response to the cost associated with changing requirements over traditional software development processes. Agile is able to embrace the changing requirements while conserving the quality of the software being developed (Highsmith & Cockburn, 2001, p.120). The backbone of agile development is derived from The Agile Manifesto (Agile Alliance, 2011):

> "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" (Agile Alliance, 2011)

All Agile methodologies adhere to these four main ideals. These ideals are supported by 12 main principles which can be found in Appendix A (Agile Alliance, 2011). The commonly used agile software processes include eXtreme Programming, and SCRUM.

### eXtreme Programming (XP)

The main factors attributed to the XP development process are customer satisfaction, teamwork, communication, simplicity, feedback and courage (Dennis et al., 2010). As a consequence of customer satisfaction, this Agile process also requires business stakeholders to be in close communication with the developers. This interaction enforces the development team to better prioritize requirements and provide feedback throughout the process (Northrop, 2004, p. 40). Furthermore the team should create simple solutions, improve design incrementally, and continuously test the solution. These tasks ensure that the cost of changing the requirements stays at a minimum (Highsmith & Cockburn, 2001, p.120).

In terms of planning, XP utilizes user stories, usually created and prioritized by the customers. The customers are also in charge of creating detailed acceptance tests for each of the features suggested in the stories. Usually the team will have their first deliverable or solution ready within the first few weeks. Each of the deliverables will be a step towards the final solution. The developers ensure that the specified requirements are met in the order they were prioritized and that the modules implemented pass the user-defined tests. The developers are also in charge of estimating the time necessary for the implementation of each feature (Martin, 2000).

### *SCRUM*

Scrum is a form of Agile development that has 3 clearly defined roles: ScrumMaster, product owner, and the development team. The scrum process involves the interaction between these three roles. At the beginning of the software project, the product owner is in charge of communicating with stakeholders of the project, outlining requirements, and adding them to the *Product Backlog*. The product backlog, a document that is a part of the SCRUM process, is a list of requirements organized by priority. The requirements for the project are gathered in the form of user stories, over the course of a one to two day requirements workshop. *Release planning* must also be done in the initial phases to determine when the software should be released over the course of the project. Once the release planning is done, the *sprint planning* comes next. A *sprint* is the time period when the software is being developed. Sprint planning is done in 2 meetings that are 4 hours each. The first meeting is to determine what the product is about and the second one is to determine how to go about creating the product (Pham, 2011).

Once the planning phase is completed, the sprint work commences. A sprint lasts between one to four weeks. Every sprint involves a sprint goal, and the progress towards this goal is inspected on a daily basis during 15 minute scrum meetings within the team. No additional requirements or bugs are added to the product backlog when the sprint is in session. It is only under a few circumstances that this can happen and must be in agreement between the team and the product owner. During the sprint there are daily meetings that can last up to 30 minutes where the team will discuss what they have accomplished towards the sprint goal. To keep track of progress, a burndown chart is generated and updated by the team during the sprint. Towards the end of every sprint, the ScrumMaster organizes a sprint review meeting for the product owner and the team. This meeting usually lasts four hours for a four week sprint and two hours for a two week sprint. The aim of the review is for the team to demonstrate to the product owners what has been done, get their feedback, and get updated changes to the product that the product owner might have (Pham, 2011).

The responsibilities of the three main roles of scrum: ScrumMaster, product owner, and the development team, can be seen in the Figure 1-3 below (Pham, 2011).

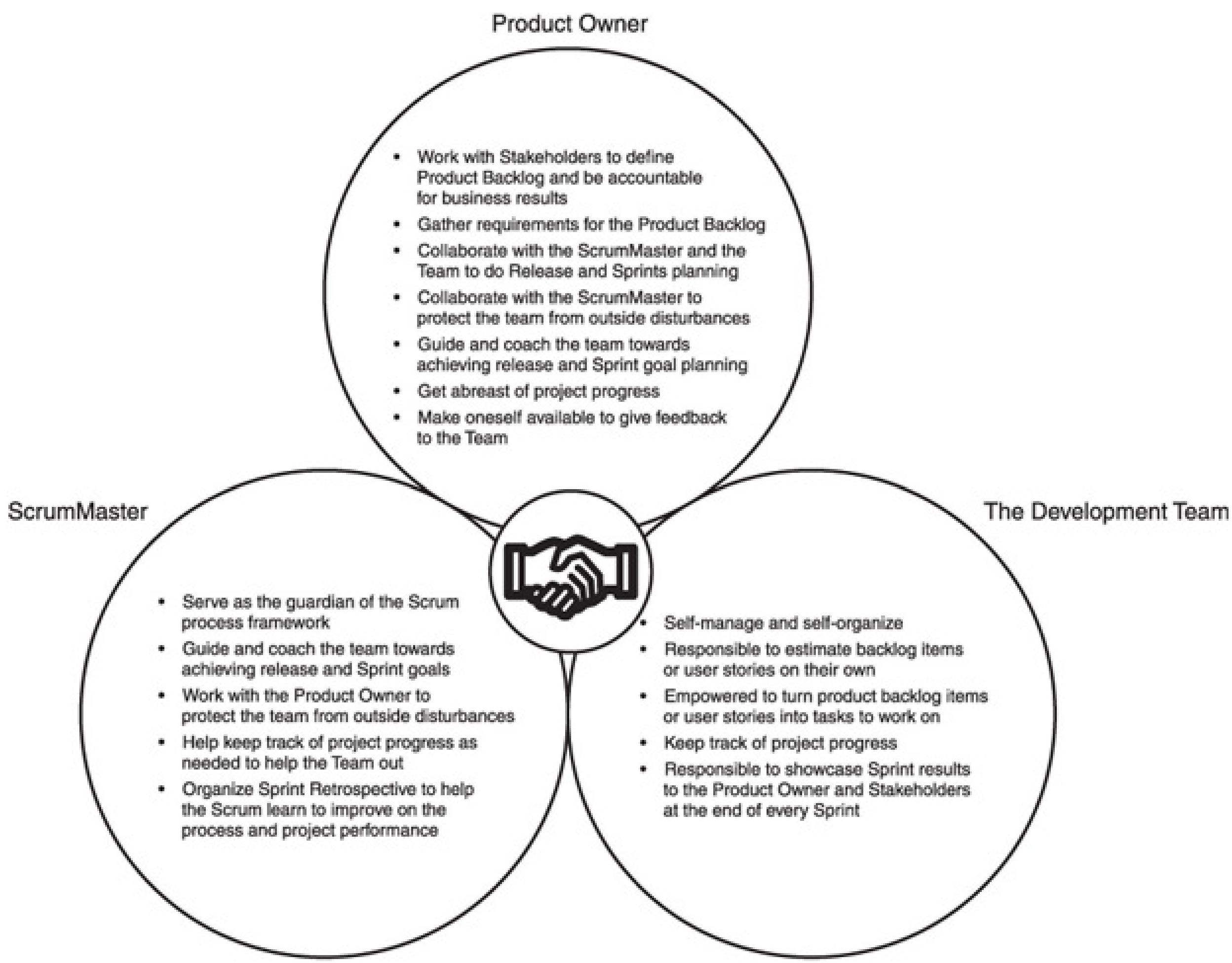


**Figure 1-3 SCRUM Diagram of Roles (Pham, 2011)**

## Decision Support System (DSS)

Since the main goal of our project is to help non-experts choose and implement a software development process, we created a system that is similar to what is generally known as a decision support system. A DSS is any computer-based system that combines knowledge from various sources and uses models to solve problems and make decisions (Power & Sharda, 2009; Sauter, 2010). There many types of DSS's, and not all of them cater to our goal of helping non-experts. This background focuses more on knowledge-driven DSSes, specifically the expert systems that help make decisions through the knowledge and skills provided by a domain expert (Chen, 2005).

## The History and Definition of DSS

The quest for a decision making system started around the 1960s in MIT as a project on the development of interactive systems. By the 1970s most developers were aware of the feasibility of implementing a DSS, and some companies had even implemented interactive systems capable of modeling data. Organizations soon recognized the need for such a system for financial, strategic decision making, and even support operations. Eventually DSS's were developed for specific organizations such as hospitals, financial institutions, and insurance companies (Power & Sharda, 2009).

The main objective behind any DSS is not to simply collect a lot of information or perform numerous analyses, but rather to collect and analyze the right *type* of information that would help users make informed decisions (Sauter, 2010). According to Power and Sharda (2009), a DSS is an interactive system that should be capable of employing "computer communications, data, documents, knowledge, and models to solve problems and make decisions" (p.1539). In another source, DSSes are also expected to "assist in the organization and analysis of information, and facilitate the evaluation of assumptions underlying the use of specific models" (Sauter, 2010, Ch1.1). Therefore, in some cases DSSes must also provide users with information on alternative decisions other than the chosen one (Sauter, 2010).

## DSS Characteristics and Components

There are a variety of opinions as to which major characteristics constitute a DSS. The following list combines the characteristics mentioned by Power and Sharda (2009) as well as Sauter (2010):

- Facilitates the decision processes
- Should support decision making and not automate it
- Must adapt to changing needs of decision-makers
- Must access data from a variety of sources
- Consolidate lots of "data" into "information" to aid decision making

These characteristics should be taken into consideration when building any DSS. Furthermore, developers must also include the decision support needs specific to organizations that they are developing for (Power & Sharda, 2009).

A typical DSS consists of four components: user interface, data management module, model management module, and the DSS architecture. The *user interface* allows a DSS user to interact and communicate with the system. Some examples of what this component could display would be dialogs, menus, icons, visual representations, and so on. The *data management module* is in charge of storing data from any source and of any form that could be useful for decision making. The models used in the system for analyzing data and providing solutions are managed by the *model management module*. Finally, the DSS *architecture* component has to do with how the three previous components are used together but kept independent to support high levels of modularity (Stanciu, 2009).

## Types of DSS - Expert Systems

Due to the large scope of industries that DSSes could support, there is naturally a variety of DSS types. *Communications-driven DSS* focuses on “communication, collaboration, and decision support” (Power & Sharda, 2009, p. 1542). The *document-driven DSS* stores and processes large amounts of data, performs appropriate analysis, and is able retrieve the relevant documents. The *model-driven DSS* does mathematical analyses and uses models pertaining to accounting, finance, representation, and optimization. Finally, the *knowledge-driven DSS* or also called the *expert system* deals with specializing in problem solving in a particular domain (Power & Sharda, 2009).

Expert systems incorporate human reasoning, expertise, and knowledge of a domain to solve problems and provide high quality results to non-experts (Chen, 2005; Sauter, 2010). The development of expert systems arrived along with the DSS around the 1960s at a few universities. Each of these expert systems had different goals. DENDRAL developed at Stanford performed analysis of chemical compound structures. At MIT, the system MACSYMA used symbolic reasoning to solve math problems. Another system also developed at MIT had an interactive dialog. Soon, researchers and industries took interest in such a system and invested in this endeavor (Chen, 2005).

When developing an expert system, one must consider several factors and characteristics. Chen (2005) states that an expert system must possess the following characteristics:

- Solves problems that requires human expertise
- Simulates human reasoning through the knowledge provided by domain experts
- Uses Heuristics and approximation methods are employed to find the best choice
- Must justify its reasoning to users who are not experts

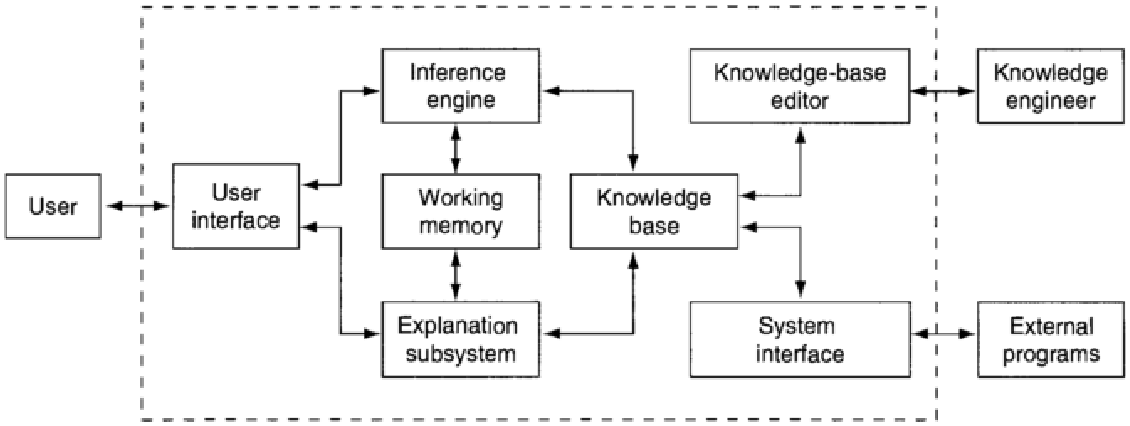


**Figure 1-4 Expert System Architecture (Chen, 2005)**

The expert system is also composed of a set of components. The two most critical are the *knowledge base* and the *inference engine*. The knowledge base is the information set aside by the domain experts as appropriate data for decision making. The inference engine deals with the heuristic or algorithm behind making the right choice and solving the problem. Figure 1-4 includes the other components of the expert system as well as a typical expert system architecture (Chen, 2005).

## Background Summary

From the research we have done on software development processes and DSSes, we were able to conclude that we will need to rely on a domain expert to make software capable of conveying expert knowledge to a non-expert. Articles we reviewed on choosing between processes agreed that ultimately, research and expertise would be necessary to choose the right process for a software project. Furthermore, the processes we chose to review are only a few of the many that currently exist. From our understanding of DSSes, it is apparent that we will not be able to put together an adequate knowledge base that would support decision making on software processes. Therefore, we decided that we would consult Professor Pollice for domain expert based information to make the Software Process Configurator possible.

# Methodology

## Problem Statement

Defining the problem and planning a solution was actually one of the most difficult and time consuming parts of this project. Professor Pollice presented us with the problem that project managers and process engineers need help researching, implementing, and managing software projects. There currently is no easy way for non-experts to find the most suitable tools and information for their software project. In addition, there are many tools to choose from, and researching them all is time consuming. When we started this project, we wanted to develop guidance software capable of helping users find the most suited software development process for their project and the relevant information necessary to implement it. We decided to call our software the Software Process Configurator (SPC). We first decided on a set of goals and requirements we hoped to achieve, which we list here.

The SPC must be capable of the following goals and requirements.

### Goals

1) Create a consolidated report that recommends processes, practices, and tools appropriate for a user's software project.

2) Provide users with the knowledge necessary to use the recommended development process.

3) Provide software process experts with a tool to edit the recommended practices and tools.

### Requirements

1) Provide users with a guided experience.

2) Determine user project specifications through a guided dialog.

3) Users must be provided all the information they need to complete the guiding questions.

4) Based on user response, the SPC should recommend the best-suited software development process.

5) The SPC will recommend the necessary tools and information to implement the suggested process.

6) Domain experts should be allowed to edit the dialog and the repository of recommended tools and information.

## User Scenarios

We devised two user scenarios to help with the designing the SPC.

### *Scenario 1*

Jake is a process engineer who is assigned to a web-based project for his company. He has to create a website to help customers manage their monthly utility bills. He does not have much experience with managing a software project. He decides to consult the SPC to find the most suitable method to manage his project. He is taken through a guided set of questions and he is able to answer them easily. He is then provided with the most suitable practice to use for his project and the recommended tools and information he needs to implement that practice. He is satisfied with the final report and is able to implement the process with all the provided information.

### *Scenario 2*

Jillian is an experienced program manager. She has tried different software development strategies for her different software projects over the years. She occasionally finds the need to do research on different software practices prior to the start of her project to ensure that the practice she is following is the ideal one. In addition, she also has to search for the appropriate tools and the best ones for every project she is assigned. This is usually a drawn out process and takes her a couple weeks to put all this information together to provide for her development team. When she is provided the SPC tool, she finds that she is able to scope her project within minutes. She likes that she is provided information on the most suitable development practice for her needs and a justification for that. In addition, she is also provided all the tools and necessary information for her project. She is satisfied that she was able to save time and get a consolidated set of information necessary to get started.

## Project Set Up

At the beginning of the project, we selected all of the tools we would use to help organize and centralize our work. We made a TeamForge project on which we would post documents, update blogs, and keep our code repository. Following Professor Pollice's recommendation, we also set up Mendeley accounts and joined his reuse group to keep track of our research sources. The blogs were our most important tool since they helped us keep track of any important research discoveries, meeting results, and code contributions.

We decided on the tools and languages we were going to use as we were going through the first design and implementation run. We wrote the SPC in Java, the language we were most comfortable with. We felt that it would allow us to spend more time on design and research than on learning a new language. For our asset database, we went with DB4o to store java objects directly without having to use SQL or a relational database. We developed the project using the Eclipse IDE, and kept a code repository on TeamForge. We also used the CodePro Analytix plug-in for Eclipse to analyze code coverage and keep our code clean and well commented. Once our development environment was set up we began design and implementation.

## Design 1 - Visualizing Dialog as a Directed Graph

Once the problem had been defined and we began our research, we started thinking about solutions to the problem. Professor Pollice recommended we look at dialog based decision making systems such as Turbo Tax. Turbo Tax is a system that helps users file their tax returns by asking the user a series of questions. After the user has answered all of the questions, Turbo Tax presents the user with a completed tax return that provides the maximum tax refund for the user. Professor Pollice envisioned a similar system that could ask the user questions about their software project and present the most appropriate development process along with useful assets to help implement the recommended process. With this solution in mind, we were able to begin think about the major components of the system.

## Design

To understand the user's project environment, we decided that we would need a dialog component. This dialog component should be in charge of issuing multiple choice questions and recording the user's answers. The dialog should also be able to adjust the questions being asked based on the answers to previous questions. Once the answers are collected from the dialog, another component would take the users responses, select the best-matched software process, and provide the most relevant resources. This component, which we called the *configurator,* would then produce a set of results to be presented to the user. The results and resources also called the *assets* are stored in a database and are accessed when the user answers are being processed.

## Implementation

The first approach we took was to represent the dialog as a directed graph approach. That is, each node corresponds to a question, and edges point to possible next questions. We identified three different types of nodes. The most important type of question node was the analysis node, which could make decisions about the next question to ask based on the previous answers. Initial dialog models were developed using Test Driven Development to help break the dialog system into small manageable pieces. Once the elements of a generic dialog system were familiar to us, we moved away from TDD in favor of faster results. A class diagram of our first dialog component can be found in Figure 2-1.

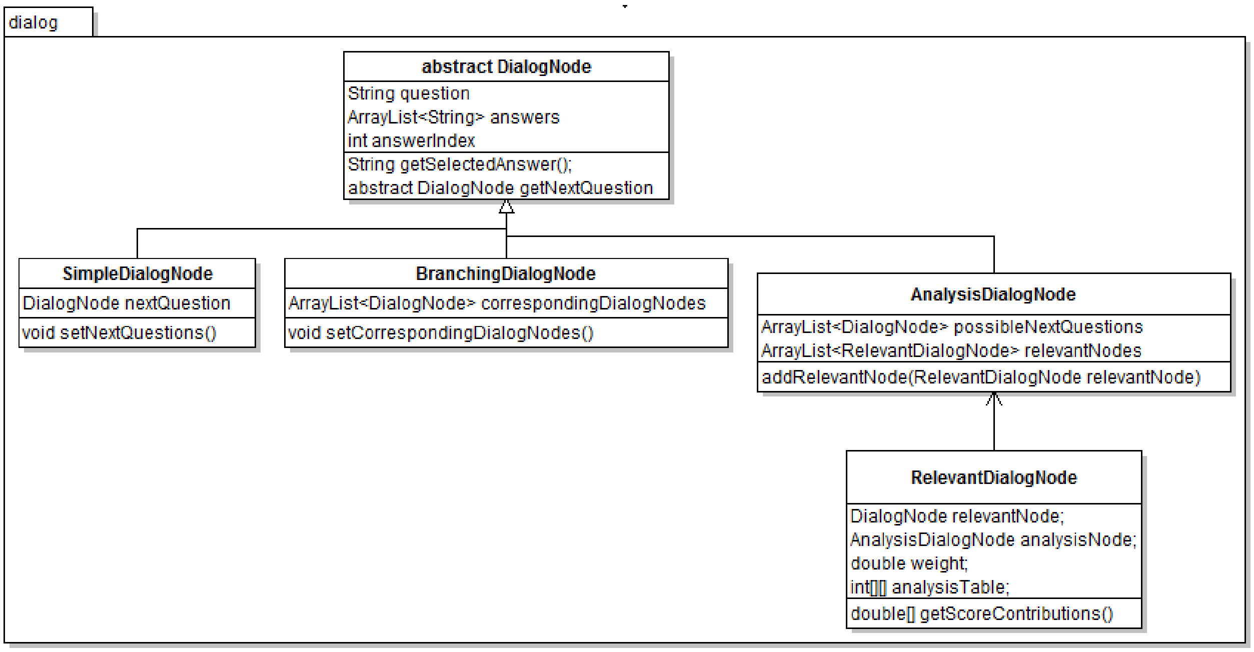


**Figure 2-1 Design 1 Dialog Package**

## Design 2 - Visualizing Dialog as a Linear list

After our initial implementation, we realized we wanted the ability to change the course of the dialog based on assets that had been recommended so far. To do this however, we needed to be able to recommend assets as the dialog proceeded. In our initial design, the configurator finishes the dialog and then recommends assets. However, conserving the dialog as a directed graph while changing the function of the configurator proved to be too cumbersome. We decided upon a second design for the dialog component and decided to implement an asset manager component rather than a configurator.

## Design

### *Dialog Manager Component*

To find an easier way to manage the dialog, we looked at a linear dialog in which questions were put in a list and asked in order. To provide the same flexibility as the directed graph approach, we devised a heuristic. Each question was given a score and a threshold that must be met for the question to be asked. After the user answers a question, that answer would add points to the score of later questions. When the later questions were reached, the dialog controller only needed to look at the questions score and threshold to determine whether or not to ask the question. This system also made the composition of the dialogs easier.

### *Asset Manager Component*

Once we were satisfied with our dialog component, we worked on a heuristic that recommends assets based on the users responses. We give each asset a numerical score and we also give scores to *tags*. A *tag* is simply a string that has a score attached to it, they work just like search tags for articles in databases. Several assets may have the same tag. Once the score of a tag is increased, it increases the chance of picking assets with that tag. Once the user is done answering the questions in the dialog, the scores of the assets and tags are summed up and the ones with the highest scores are chosen to be displayed.

## Design 3 - Final Architecture

Although Design 2 proved to be promising we altered the dialog component to better suit our needs. We included the idea of assets and tags determining the next question in the dialog. We found that it might be difficult for the dialog writer managing scores and thresholds to determine how the questions would play out.

## Dialog

We continue to use the linear dialog; however we decided to implement two additional concepts: recommendations and triggers. A recommendation is an action that is run when a user selects an answer for a question in the dialog. These actions include adding points to an asset or tag score. A trigger is an optional part of every question that adds a condition to the question. Every element of the condition must be met before presenting the question. These condition elements could be specific user responses from other questions, assets/tags that have already been selected for the result set, or assets/tags with a certain score. Furthermore, the condition elements can be combined with the

Boolean operators “and”, “or”, and “not”. For example, in the trigger for question 6 in a dialog, the condition could explicitly state that answer A from question 1 must not have been chosen for question 6 to display.

Both the recommendation and trigger features take care of asset scoring and the problem of determining the next question from the dialog to display. A UML class diagram of these features can be seen in Figure 2-2. We also include a sequence diagram that demonstrates the question asking process in Figure 2-3 .

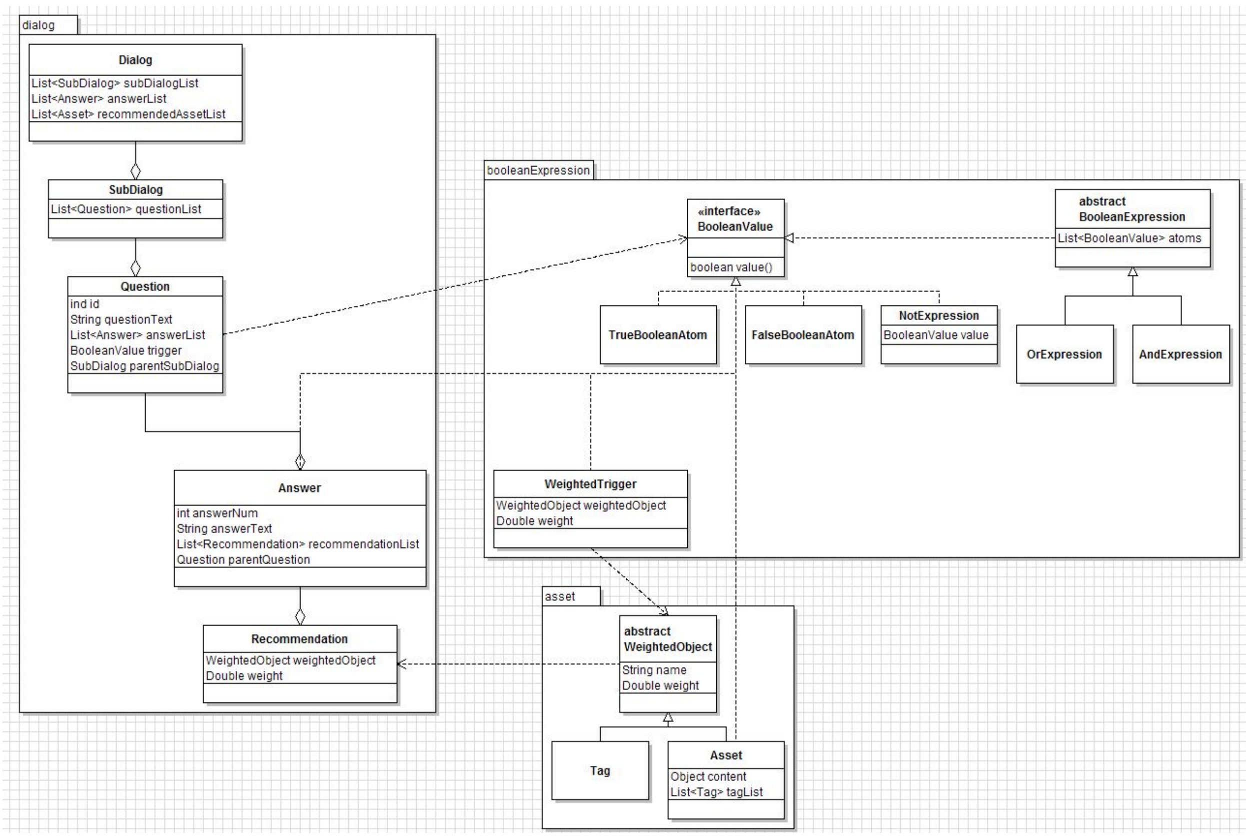

**Figure 2-2 Final UML Diagram**

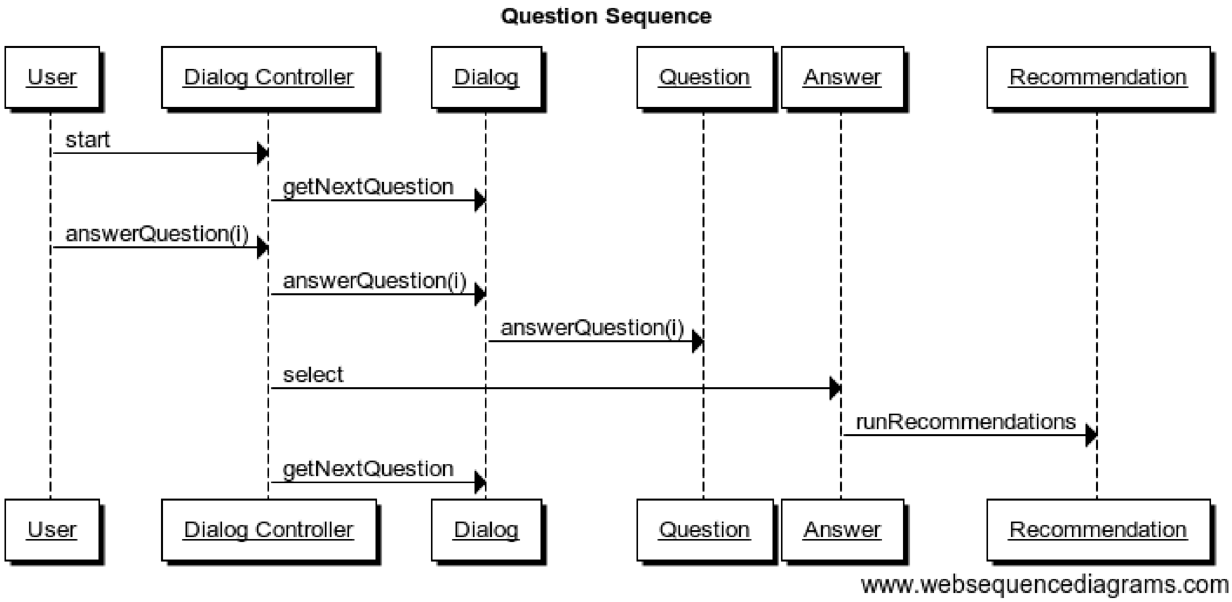


**Figure 2-3 Next Question Sequence Diagram**

## Dialog Writing

Since we had the dialog and asset management structures in order, we included a way to input a dialog written by a software process expert into the SPC. We took Professor Pollice's recommendation and chose to use XML as the form of input. We felt that XML would be an easy way to represent all the components needed to create a dialog. To turn an XML input dialog into a dialog object that is capable of being run by the SPC, we use a dialog parser. As we tried writing sample dialogs, we soon realized that dialogs quickly became too long. The recommendation and trigger dependencies also became confusing with larger dialogs. To solve this problem, we introduced the concept of subdialogs. These subdialogs would also help the dialog writer focus on smaller tasks and break down the decision making process into smaller, more manageable pieces. We allowed the triggers that are applied to questions to also be used with a subdialog, where the conditions in it are checked before asking the questions in the subdialog. To manage the order in which these subdialogs were presented we decided that there must be an "index" file to specify the order.

### *Sample Dialog Structure*

The sample dialog structure below shows how a larger requirements dialog can be split up into smaller subdialogs that deal with fewer triggers and recommendations. For a more explicit tutorial on how to write the dialog in XML, refer to Appendix C.

Subdialog A - Finds out who is in charge of the requirements and if they already have been determined.

> *Recommends either*: “Tag: Client determines requirements”, “Tag: Team determines requirements”, “Tag: Have requirements”, or “Tag: Find requirements”

Subdialog B - Helps user find the appropriate requirements type.

> *Triggers:* (“Tag: Find requirements”), (“Client determines requirements” and “Team determines requirements”), or (Answer a in question 5 of Subdialog A)
>
> *Recommends:* “Tag: Informal”, “Tag: Formal”, “Asset: Use cases”, “Asset: User Scenarios”, “Asset: User Stories”, “Asset: SRS”, or “Asset: Technical Specifications”

Subdialog C - Helps user find format appropriate for requirements type.

> *Triggers:* (“Tag: Client determines requirements” or “Tag: Team determines requirements”), or (“Asset: Use cases”, “Asset: User Scenarios”, “Asset: SRS”),
>
> Not (“Answer c in question 2 of Subdialog B”)
>
> *Recommends:* “Asset: Use cases F1”, “Asset: Use cases F2”, “Asset: User Scenarios F1”, “Asset: User Scenarios F2”, “Asset: SRS F1”, or “Asset: SRS F2”

## Graphical User Interface (GUI)

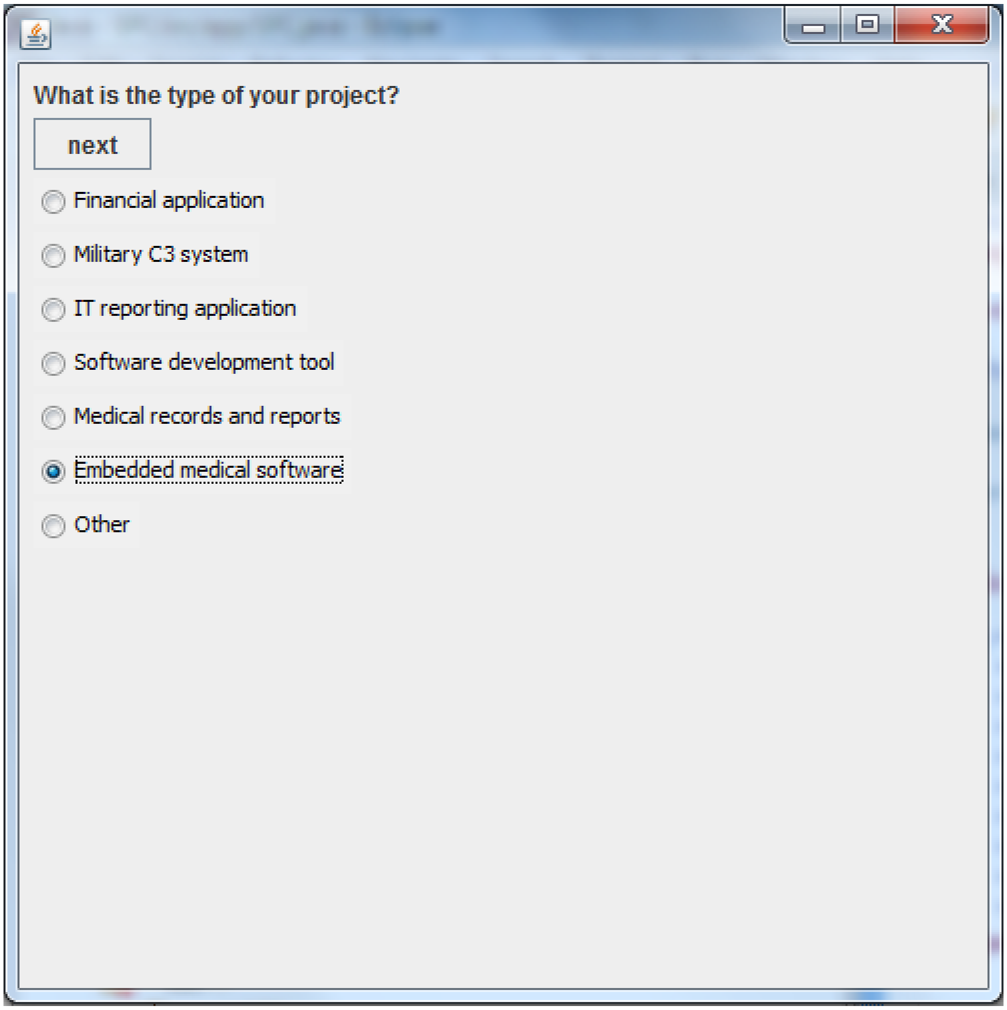


Figure 2-4 Dialog GUI

We also created a very simple GUI (Figure 2-4) that navigates the questions in the dialog and records the user responses. Once the user completes the dialog, the console displays the set of most suited assets and/or tags.

## Results

Through the SPC we showed that a guided dialog written by an expert can be used to understand a user's software project environment. We also ensured that the dialog was dynamic and that it was not just a linear set of questions. The dialog writer/domain expert is also given the flexibility to write and configure the dialog such that the users of the SPC are presented with the most relevant questions to their situation. Finally, we also provide dialog writers the ability to provide a heuristic for assets that the dialog is configured to pick out as a user is answering the questions in the dialog. Additionally, non-experts such as process engineers can be guided through the expert knowledge base easily to find the most relevant information on software development for their scenario.

Currently, our user interface is very simple and is capable of traversing the questions in the dialog. The console displays the list of best matched assets once the user is done with the dialog. Also, our dialog input method is specifically in XML. Although ideally we would have liked to include a method to input assets into the SPC, our advisor mentioned that it was out of scope and that it would consume too much of the time allotted for this project. We will discuss specific areas of the SPC that can be improved later, in the Future Work section.

Our advisor, Professor Pollice, was satisfied that we were able to devise a proof of concept that knowledge and information provided by experts can be reused through the SPC. Our work through design and implementation of our different dialog and asset models made us reach a flexible strategy for this concept. We also discovered that as we were creating our SPC, we were creating an interface that anybody with a decision making strategy could use. For example, we created a dialog called "Where should I eat?" which asks a user a series simple of questions about hunger, thirst, and where they are located. The result page provides the user with the information on where they could go and how they could get there. The XML dialog for this can be found in Appendix D. This flexibility would provide future developers who want to use the SPC framework, with the ability to customize it for any decision making process.

## Future work

The SPC is open to a lot of work in terms of testing its functionality in real life scenarios. The categories below explain the aspects of SPC that we think could be improved or used for future work.

### Dealing with Dialog Errors

We are not testing any of the XML dialog written by a domain expert. If the SPC were to be applied in a real life scenario, the dialog should be checked for dead ends, paths which don't recommend assets and questions that are never asked.

In addition to logic based errors, there could also be syntactical errors in the XML provided by the dialog expert. One such example is misspelled tags, those are usually ignored, and the dialog expert is never notified of it. The SPC could use an extensive amount of error checking for bad input. We assume that the XML Dialogs that are provided by the domain expert are in the perfect form. This leads to the next idea of changing input methods for the dialog.

### Future Forms of Dialog Inputs

Writing dialogs in XML may not be the most aesthetic or most natural way for human experts. An easier to view dialog editor interface would help domain experts to worry less about the XML syntax and focus better on the dialog structure.

### Asset Database Editor

We do not have a user interface for input of assets into the SPC. Ideally, when a domain expert is writing the dialog, they will also be supplying the assets for the database since they are specifically writing the dialog to choose between the assets in their domain.

## Results Templates

As of now, the Results are displayed in the console in text form. Only the title of the asset or tag is actually being displayed. The quality of the results being produced by the SPC could be increased if the result set was required to follow a template written by the dialog author. For example, we would like the asset describing the most suitable process to be a mandatory asset displayed on the results template. The template could indicate which tags need to be covered and how many assets are needed for each particular area. For example, the dialog writer could specify that they would like 2 highest ranked assets with tag x, and then the highest scoring asset with tag y on the template. This would allow the results to be displayed in an order that makes sense to the user as opposed to just a list that may or may not be sorted by order of score.

The template could also be editable by the dialog itself. This would allow the dialog to change the number or type of assets that are recommended based on the user's answers. For example, the standard template may ask for 1 asset on writing user stories, but if the user indicates that they have no experience with user stories, the dialog writer may want to change that and give them 2 or 3 assets.

## Asset Categorization

Currently all assets are not categorized by what kind of resource they are. A book is the same as a website which is the same as a tool. If assets were given types, even more strategies could be made in terms of recommending assets, and templates may be configured to display one resource of "book" type or "tool" type. This could be done already by an ambitious dialog writer by adding tags like "book" or "article" to Assets.

## Complex Recommendations

Recommendations are all simple statements. They are able to add points to any asset or tag. Dialog writers may want the flexibility to pick and choose assets, for example, they may only want to choose assets that are tagged *y* and are not tagged for *z*. If the same Boolean logic system that triggers use was applied to recommendations, dialog authors could write much more specific recommendations.

## Testing for Proper Results

Any dialog composed by a domain expert should be tested using a real life scenario to see if appropriate results are being reached. The domain expert can outline a scenario and give his or her opinion on what decision should be made in that scenario. This scenario can then be run through the SPC to see if the resulting assets come close to the decision that the domain expert makes.

There are many potential uses for the SPC and many improvements to be made for use in real life scenarios. However, the SPC framework should provide an adequate starting point for decision making software that requires knowledge bases from domain experts.

## Works Cited


Alexander, L. C., & Davis, A. M. (1991). Criteria for selecting software process models. *[1991] Proceedings The Fifteenth Annual International Computer Software & Applications Conference* (pp. 521-528). IEEE Comput. Soc. Press. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=170231

Baskerville, R., & Pries-Heje, J. (2004). Short cycle time systems development. *Information Systems Journal*, *14*(3), 237-264. Retrieved from http://au4sb9ax7m.search.serialssolutions.com/?url_ver=Z39.88-2004&rfr_id=info:sid/IEEE.org:XPLORE&rft_id=info:doi/10.1111/j.1365-2575.2004.00171.x&url_ctx_fmt=info:ofi/fmt:kev:mtx:ctx&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&rft.jtitle=Information Systems Journal&rft.date=2004&rft.volume=14&rft.issue=3&rft.spage=237&rft.issn=13501917&rft.au=Baskerville, Richard Pries-Heje, Jan&rft.aulast=Baskerville, Richard Pries-Heje, Jan

Beck, K. (2002). *Test Driven Development: By Example*. Retrieved from http://proquest.safaribooksonline.com.ezproxy.wpi.edu/book/software-engineering-and-development/software-testing/0321146530/firstchapter#X2ludGVybmFsX0ZsYXNoUmVhZGVyP3htbGlkPTAtMzIxLTE0NjUzLTAvcGFydDAx

Beck, K. (n.d.). *Extreme Programming Explained: Embrace Change*. Addison-Wesley. Retrieved from http://books.google.com/books?hl=en&lr=&id=G8EL4H4vf7UC&oi=fnd&pg=PR13&dq=extreme+programming+small+teams&ots=j8uDypkWup&sig=e7yNuEtZJ7uJ3lTaYr3NfZryyDY#v=onepage&q=extreme programming small teams&f=false

Bildhauer, D., Horn, T., & Ebert, J. (2009). Similarity-driven software reuse. *2009 ICSE Workshop on Comparison and Versioning of Software Models*, 31-36. Ieee. doi:10.1109/CVSM.2009.5071719

Burstein, F., & Holsapple, C. F. (2008). Decision support systems in context. *Information Systems and E-Business Management*, *6*(3), 221-223. Retrieved from http://dx.doi.org/10.1007/s10257-008-0085-1

Chen, W.-K. (2005). Expert Systems. *Electrical Engineering Handbook*. Elsevier. Retrieved from http://www.knovel.com/web/portal/browse/display?_EXT_KNOVEL_DISPLAY_bookid=1713&VerticalID=0

Cockburn, A. (2000). Selecting a project's methodology. *IEEE Software*, *17*(4), 64-71. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=854070

Cohn, T., Blunsom, P., & Goldwater, S. (2010). Inducing tree-substitution grammars. *The Journal of Machine Learning Research*, *9999*, 3053–3096. JMLR. org. Retrieved from http://portal.acm.org/citation.cfm?id=1953031

Cusumano, M. A., MacCormack, A., Kemerer, C. F., & Crandall, W. (Bill). (2009). Critical Decisions in Software Development: Updating the State of the Practice. *IEEE Software*, *26*(5), 84-87. Cambridge. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5222801

Davies, R. (2001, April). The power of stories. *WWW Retrieved*. Retrieved from http://www.agilexp.com/presentations/PowerOfStories.pdf

Dennis, A., Wixom, B. H., & Roth, R. M. (2010). *Systems Analysis and Design*. John Wiley & Sons, Inc.

Erdogmus, H. (2008). Essentials of Software Process. *IEEE Software*, *25*(4), 4-7. Retrieved from http://ieeexplore.ieee.org.ezproxy.wpi.edu/stamp/stamp.jsp?tp=&arnumber=4548398

Fayad, M. E. (1997). Software Development Process: The Necessary Evil? *Communications of the ACM*. Retrieved from http://www.engr.sjsu.edu/fayad/publications/columns/Column1.html

Feiler, P. H., & Humphrey, W. S. (1993). Software process development and enactment: concepts and definitions. *Second International Conference on the Software Process, 1993. Continuous Software Process Improvement* (pp. 28-40). Pittsburgh. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=236824

Gary, L. (2005). Expert Systems. *Expert Systems*.

Henninger, S. (1997). An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology*, *6*(2), 111-140. doi:10.1145/248233.248242

Henninger, S., & Schlabach, J. (2001). A tool for managing software development knowledge. *Product Focused Software Process Improvement*, 182–195. Springer. Retrieved from http://www.springerlink.com/index/4a8wblam4q377257.pdf

Highsmith, J., & Cockburn, A. (2001). Agile software development: the business of innovation. *Computer*, *34*(9), 120-127. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=947100

Hsieh, M. (2006). Supporting software reuse by the individual programmer. *ACSC ’06 Proceedings of the 29th Australasian Computer Science Conference* (Vol. 48). Retrieved from http://portal.acm.org/citation.cfm?id=1151702

Hull, M. E. ., Taylor, P. ., Hanna, J. R. ., & Millar, R. . (2002). Software development processes — an assessment. *Information and Software Technology*, *44*(1), 1-12. Retrieved from http://au4sb9ax7m.search.serialssolutions.com/?ctx_ver=Z39.88-2004&ctx_enc=info:ofi/enc:UTF-8&rfr_id=info:sid/summon.serialssolutions.com&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&rft.genre=article&rft.atitle=Software+development+processes+-+an+assessment&rft.jtitle=INFORMATION+AND+SOFTWARE+TECHNOLOGY&rft.au=Hull,+MEC&rft.au=Taylor,+PS&rft.au=Hanna,+JRP&rft.au=Millar,+RJ&rft.date=2002-01-15&rft.pub=ELSEVIER+SCIENCE+BV&rft.issn=0950-5849&rft.volume=44&rft.issue=1&rft.spage=1&rft.epag

IBM Rational Method Composer: Part 1: Key concepts. (2005, December 15). Retrieved from http://www.ibm.com/developerworks/rational/library/dec05/haumer/

IEEE. (2004). *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Angela, Burgess. Retrieved from http://www.computer.org/portal/web/swebok/html/copyright

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*. Reading, Massachusetts: Addison Wesley Longman, Inc.

Jalote, P. (2005). *An Integrated Approach to Software Engineering*. (D. Gries & F. B. Schneider, Eds.) (3rd ed., pp. 1-77). Springer Science-i-Business Media, Inc. Retrieved from http://www.springerlink.com.ezproxy.wpi.edu/content/v6g0427128833k7j/

Kim, Y., Stohr, E. A., & others. (1997). Sofhvare Reuse: Survey and Research Directions. *Business*. Citeseer. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.41

Kotonya, G., & Sommerville, I. (1998). Requirements Engineering: An introduction to requirements engineering. Retrieved from http://www.computing.dcu.ie/~nbrophy/ca222/week4_6/som1.pdf

Krueger, C. W. (1992). Software Reusse. *ACM Computing Surveys*, *24*(2), 131-83.

Li, E. Y. (1990). Software Testing In A System Development Process : A Life Cycle Perspective. *Journal of Systems Management*, *41*(August), 23-31.

Lindvall, M., & Rus, I. (2000). Process diversity in software development. *IEEE Software*, *17*(4), 14-18. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=854063

Mahoney, M. S. (2004). Finding a History for Software Engineering. *Annals of the History of Computing, IEEE*, *26*(1), 8-19. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1278847

Martin, R. C. (2000). eXtreme Programming development through dialog. *IEEE Software*, *17*(4), 12-13. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=854062

Matthews, C. (1999). Using formal grammars to encode expert problem solving knowledge. *Proceedings of the 37th annual Southeast regional conference (CD-ROM) on - ACM-SE 37*, 29-es. New York, New York, USA: ACM Press. doi:10.1145/306363.306400

McDermott, J. (1980). *R1: A Rule-Based Configurer of Computer Systems*. Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA223957&Location=U2&doc=GetTRDoc.pdf

Merriam-Webster. (2012). process. Retrieved from http://www.merriam-webster.com/dictionary/process

Miller, G. (2003). Want a Better Software Development Process ? Complement It. *IT Professional*, *5*(5), 49-51. Retrieved from http://ieeexplore.ieee.org.ezproxy.wpi.edu/xpls/abs_all.jsp?arnumber=1235610&tag=1

Nguyen, T. N. (2006). A decision model for managing software development projects. *Information & Management*, *43*(1), 63-75. Retrieved from http://au4sb9ax7m.search.serialssolutions.com/?ctx_ver=Z39.88-2004&ctx_enc=info:ofi/enc:UTF-8&rfr_id=info:sid/summon.serialssolutions.com&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&rft.genre=article&rft.atitle=Software+development+processes+-+an+assessment&rft.jtitle=INFORMATION+AND+SOFTWARE+TECHNOLOGY&rft.au=Hull,+MEC&rft.au=Taylor,+PS&rft.au=Hanna,+JRP&rft.au=Millar,+RJ&rft.date=2002-01-15&rft.pub=ELSEVIER+SCIENCE+BV&rft.issn=0950-5849&rft.volume=44&rft.issue=1&rft.spage=1&rft.epag

Northrop, R. (2004). The Fall of Waterfall. *Intelligent Enterprise*, 40.

Norton, R. J. (2003). *REUSE OF PERSONAL SOFTWARE ASSETS*. *System*. Citeseer. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.1557&rep=rep1&type=pdf

O’Regan, G. (2006). Cleanroom and Software Reliability. *Mathematical approaches to software quality* (pp. 176-196). London: Springer London.

Pemberton, S. (n.d.). Executable Semantic Definition of Programming Languages Using Two-level Grammars (Van Wijngaarden Grammars). Retrieved June 8, 2011, from http://homepages.cwi.nl/~steven/vw.html

Petersen, K., Wohlin, C., & Baca, D. (2009). *The Waterfall Model in Large-Scale* (pp. 386-400).

Pham, A. (2011). *Scrum® in Action: Agile Software Project Management and Development*. Course Technology PTR. Retrieved from http://library.books24x7.com/toc.aspx?bookid=40464

Power, D. J., & Sharda, R. (2009). Decision Support Systems. In S. Y. Nof (Ed.), *Springer Handbook of Automation* (pp. 1539-1548). Springer-Verlag Berlin Heidelberg. Retrieved from http://www.springerlink.com/content/nq0687v416k21076/

Prieto-Díaz, R. (1991). Making software reuse work: an implementation model. *ACM SIGSOFT Software Engineering Notes*, *16*(3), 61-68. doi:10.1145/127099.127107

Ramachandran, M. (2005). Software Reuse Guidelines. *ACM SIGSOFT Software Engineering Notes*, *30*(3), 1–8. ACM. Retrieved from http://portal.acm.org/citation.cfm?id=1061889

Rising, L., & Janoff, N. S. (2000). The Scrum software development process for small teams. *IEEE Software*, *17*(4), 26-32. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=854065

Sametinger, J. (1997). Software Engineering with Reusable Components.

Sauter, V. L. (2010). *Decision Support Systems for Business Intelligence, Second Edition* (2nd ed.). John Wiley & Sons. Retrieved from http://proquest.safaribooksonline.com.ezproxy.wpi.edu/book/databases/business-intelligence/9780470433744

Schwaber, K. (2004). *Agile Project Management with Scrum*. (R. Van Steenburgh, L. Engelman, & K. Atkins, Eds.) (pp. 1-17). Redmond: Microsoft Press. Retrieved from http://www.bjla.dk/VideregUdvikling/DM052/ScrumProjectManagementPart00.pdf

Srinath, S., Venkatesh, K., & Ram, D. J. (1997). An integrated solution based approach to software development using unified reuse artifacts. *ACM SIGSOFT Software Engineering Notes*, *22*(4), 56-60. doi:10.1145/263244.263258

Stanciu, C. O. (2009). Decision Support Systems Architectures. *Annals. Computer Science Series.*, *7*(2009), 341-348. Retrieved from http://arxiv.org/ftp/arxiv/papers/0906/0906.0863.pdf

The Agile Manifesto. (2011).*Agile Alliance*. Retrieved 2011, from http://www.agilealliance.org/the-alliance/the-agile-manifesto/

The Twelve Principles of Agile Software . (2011).*Agile Alliance*. Retrieved 2011, from http://www.agilealliance.org/the-alliance/the-agile-manifesto/the-twelve-principles-of-agile-software/

Tool, S., Data, Q., Company, A., Software, R., Concept, C., & March, C. (2005). *Certification Authorities Software Team ( CAST ) Position Paper*.

Zwass, V. (2012). expert system. *Encyclopædia Britannica*. Retrieved from http://www.britannica.com.ezproxy.wpi.edu/EBchecked/topic/198506/expert-system

## Appendix A – Twelve Principles of Agile

These are the twelve principles as defined on the agile alliance website (Agile Alliance, 2011):

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

# Appendix B – How the code works – Developers Guide

## Overview

The goal of the SPC is to store a set of Assets that may or may not be useful to a user, and rank those Assets according to usefulness based on the user's responses to a set of questions. To do this, we need an Asset database along with a way to modify its contents, and a way to write and administer questions to the user. We also need some method of scoring assets as answers are provided by the user, and a way to return these results to the user.

## The Asset Database

The SPC uses db4o, an object database, to store Assets, and Tags. An Asset is really anything that could be recommended by a domain expert to user. It could be anything from a website to a book to a software tool. The Asset object in the SPC holds a pointer to its content, as well as a numeric score, and a list of Tags. The score represents how useful the Asset is to the user, and is updated by the SPC as the user answers questions. A Tag is simply a string and a numeric score. A Tag's score can also be updated by SPC. Tags are useful because the same Tag can be given to multiple Assets, allowing all of those Assets to receive a score increase as opposed to having to increase the score of each Asset individually. We could have chosen to not store a score for each Tag and instead just update the Asset's score when a Tag match occurs, but keeping a separate score for the Asset as well as all of its tags allows for more flexible strategies for selecting the best Assets in the end. One strategy may consider the Asset's score more strongly than its Tag's scores, while another may look only at an Asset's Tag's scores.

## The Dialog

At the center of the SPC is the Dialog package. This package holds all of the classes that collectively make up a Dialog for the SPC. The Dialog class itself if a singleton, since we will only be dealing with one Dialog at a time for the foreseeable future. This also allows us to make static calls to the Dialog class, and gives quick and easy access to the same single Dialog object anywhere in the code.

The Dialog class is also responsible for maintaining progress through itself. That is, it is responsible for knowing which question the user is on, and is able to select the next appropriate question based on the user's responses so far.

The Dialog package also includes the SubDialog, Question and Answer classes. This section will also cover the Recommendation class even though it is in the asset package because it is an important part of a Dialog. The Dialog class has a list of SubDialog, while the SubDialog class has a list of Question, the Question class a list of Answer, and the Answer class a list of Recommendation. By giving the Dialog writer the ability to write multiple SubDialogs as opposed to one large Dialog, we encourage small, focused SubDialogs, which will hopefully yield a more organized Dialog as a whole. The SubDialog class is also important because it gives the SPC the ability to skip over entire groups of questions that may not be relevant based on a user's answers. How this works will be covered more in the trigger part of this section.

Question's and Answer's are exactly what they appear to be. For each Question created, there is a list of possible Answers to the Question. For the time, SPC Dialogs are limited to asking multiple choice type questions. An important thing to notice about the Answer class is that an Answer knows how to select itself. That is, when the user chooses an Answer, that Answer object is responsible for marking itself as selected, and running all of its Recommendations.

A Recommendation represents a change to the results of the SPC. A Recommendation is really an interface, and is implemented by WeightedRecommendation and Asset. If an Asset is recommended, it is simply added to the results list. If a WeightedRecommendation is made, the WeightedObject in the WeightedRecommendation (either an Asset or a Tag) is incremented by a specified amount. Each Recommendation knows how to apply itself, and also how undo itself. So, when an Answer is selected, it runs all of its Recommendations immediately. An Answer also knows how to deselect itself, in which the Answer undoes all of its Recommendations.

## Inputs

The SPC takes Dialog input from a domain expert in the form of XML files. This input type gives the writer total control over the Dialog. The writer may also choose to take advantage of XML editing tools to make writing the Dialog easier.

The SPC is built to handle other, possibly more glamorous forms of input that may be developed in the future. Any input type could be supported, so long as it can parse its input into the Dialog objects that the SPC uses.

## The Trigger System

The Trigger system for the SPC allows irrelevant questions to be skipped over based on information that has already been provided by the user. Questions and SubDialogs have Triggers, and may be deemed irrelevant and skipped if a previous question was answered a certain way, a certain asset has already been selected, a certain asset or tag has already reached a certain weight, or any combination of these.

The trigger system uses principles from Boolean logic to combine different types of conditions. Conditions may be combined using nested AND, OR, and NOT statements to create any condition desired regardless of how complex.

When an Answer is selected by the user, the Dialog class finds the next appropriate Question to ask. It does this by looking at the Triggers of the next Questions in line. If the Trigger for the following Question is not met, it is skipped, and the Question after that is considered. This process continues until a Trigger is met or until a Question without a Trigger is reached.

# Appendix C – XML Dialog Tutorial

A Dialog for the SPC consists of one index file and one or more SubDialog files. The SPC can create a single dialog from multiple xml files by reading the index file and merging all of the SubDialog files. This document will explain how to write an index file, SubDialog files, and how to take advantage of Triggers and Recommendations. This tutorial assumes familiarity with XML syntax.

## Index file

The index file simply lists all of the SubDialogs that should be included in the Dialog. Here is an example of an index file that lists two SubDialog files.

```
<?xml version="1.0" encoding="UTF-8"?>
<Dialog>

      <SubDialog name="setupSubDialog"></SubDialog>
      <SubDialog name="requirementsTypeSubDialog"></SubDialog>

</Dialog>
```

## SubDialog Basics

Every SubDialog follows the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<SubDialog title="SubDialogName">

      <Question id="1" questionStatement="Question 1 here?">
      </Question>

      <Question id="2" questionStatement="Question 2 here?">
      </Question>

</SubDialog>
```

The <SubDialog> tag indicates the beginning and end of the SubDialog. The SubDialog tag has one attribute, which is the title.

Each SubDialog contains questions. Add a Question using the <Question> tag. The Question tag has two attirbutes, the id, which corresponds to the questions placement in the SubDialog, and the questionStatement, which is the Question itself.

Every Question can contain multiple Answers. A Question with Answers looks like this…

```
<Question id="3" questionStatement="What kind of meal do you want?">
      <Answer text="answer 1"></Answer>
      <Answer text="answer 2"></Answer>
      <Answer text="answer 2"></Answer>
</Question>
```

## Recommendations

Every Answer may contain one or more Recommendations. Recommendations are actions that are run when a certain Answer is selected. A Recommendation may add points to a Tag or Asset directly, or recommend an Asset without adding points to it. An Answer with Recommendations looks like this:

```
<Answer text="Yes">
      <Recommendation type="tag" name="t1" weight="5"></Recommendation>
      <Recommendation type="asset" name="a1" weight="3"></Recommendation>
</Answer>
```

An answer with an Asset Recommendation without a weight looks like this…

```
<Answer text="Yes">
      <Recommendation type="asset" name="a1"></Recommendation>
</Answer>
```

This type of Recommendation automatically adds the Asset to the list of results instead of adding points to it, which may or may not lead to the Asset being recommended.

## Triggers

Questions may also contain a Trigger. A trigger is a condition that must be met in order for the question to be asked. One type of trigger is called an answer Trigger. An answer Trigger is met when the Answer in the Trigger has been selected by the user in one of the previous Questions. A Question with an answer Trigger looks like this:

```
<Question id="3" questionStatement="What kind of meal do you want?">
      <Answer text="answer 1"></Answer>
      <Answer text="answer 2"></Answer>
      <Answer text="answer 2"></Answer>
      <Trigger>
            <question question="1" answer="2"></question>
      </Trigger>
</Question>
```

The Trigger must be given the Question id and the Answer number of the answer that the condition relies on. An answer Trigger can also reference answers in other SubDialogs by including the name of the other SubDialog. This kind of Trigger looks like this…

```
<Trigger>
      <question references="name" question="1" answer="2"></question>
</Trigger>
```

An additional type of Trigger is an asset Trigger. This kind of Trigger is met if the Asset given has already been selected for the user. This type of Trigger looks like this:

```
<Trigger>
      <asset name="asset name"></asset>
</Trigger>
```

Weighted Triggers are Triggers that are met if the specified Asset or Tag has reached a given weight. These Triggers look like this:

```
<Trigger>
	<asset name="City bar guide" weight="2"></asset>
</Trigger>
```

In addition to these Triggers, you can also use common expressions from boolean logic like and, or, and not. A not Trigger is simply any Trigger surrounded by <not> tags. A not Trigger looks like this…

```
<Trigger>
	<not>
		<question question="3" answer="1"></question>
	</not>
</Trigger>
```

And Triggers and or Triggers can conain mulptiple Triggers. These Triggers look like this…

```
<Trigger>
	<and>
		<question question="3" answer="4"></question>
		<question question="4" answer="2"></question>
	</and>
</Trigger>
```

This and Trigger will be triggered if question 3 answer 4 has been selected AND question 4 answer 2 has been selected.

```
<Trigger>
	<or>
		<question question="5" answer="2"></question>
		<question question="4" answer="2"></question>
	</or>
</Trigger>
```

This OrTrigger will be triggered if question 5 answer 2 has been selected OR question 4 answer 2 has been selected.

You may nest these boolean expressions to make more complex Triggers like…

```
<Trigger>
	<and>
		<or>
			<question question="5" answer="2"></question>
			<question question="4" answer="2"></question>
		</or>
		<not>
			<question question="4" answer="1"></question>
		</not>
	</and>
</Trigger>
```

Triggers can also be applied to SubDialogs. To apply a trigger to a SubDialog, simply add the Trigger clause to the SubDialog like this…

```
<?xml version="1.0" encoding="UTF-8"?>
<SubDialog title="testSubDialog2">

	<Trigger>
		<asset name="a1"></asset>
	</Trigger>

	<Question id="1" questionStatement="q1">
		<Answer text="a1"></Answer>
		<Answer text="a2"></Answer>
	</Question>
</SubDialog>
```

## Appendix D – "Where should I eat?" Test Dialog

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SubDialog title="testSubDialog">

      <Question id="1" questionStatement="Are you hungry or thirsty?">
            <Answer text="Hungry" asset="t1" weight="2"></Answer>
            <Answer text="Thirsty"></Answer>
      </Question>

      <Question id="2" questionStatement="Are you 21+">
            <Answer text="Yes">
                  <Recommendation type="tag" name="21+"
weight="5"></Recommendation>
                  <Recommendation type="asset" name="City bar guide"
weight="3.0"></Recommendation>
            </Answer>
            <Answer text="No"></Answer>

            <Trigger>
                  <question question="1" answer="2"></question>
            </Trigger>
      </Question>

      <Question id="3" questionStatement="What kind of meal do you want?">
            <Answer text="A quick snack"></Answer>
            <Answer text="Fast food"></Answer>
            <Answer text="Casual lunch"></Answer>
            <Answer text="Formal dinner"></Answer>
            <Trigger>
                  <asset name="City bar guide"></asset>
            </Trigger>
       </Question>

      <Question id="4" questionStatement="Do you have transportation?">
            <Answer text="Yes"></Answer>
            <Answer text="No"></Answer>
            <Trigger>
                  <not>
                        <question question="3" answer="1"></question>
                  </not>
            </Trigger>
      </Question>

      <Question id="5" questionStatement="Would you like to hire a limousine
for the evening?">
            <Answer text="Yes"></Answer>
            <Answer text="No"></Answer>
            <Trigger>
                  <and>
                        <question question="3" answer="4"></question>
                        <question question="4" answer="2"></question>
                  </and>
            </Trigger>
      </Question>
```

```
	<Question id="6" questionStatement="Are you familiar with the local
establishments?">
		<Answer text="Yes"></Answer>
		<Answer text="No">
			<Recommendation type="asset" name="City bar guide"
weight="2.0"></Recommendation>
		</Answer>
		<Trigger>
			<or>
				<question question="5" answer="2"></question>
				<question question="4" answer="2"></question>
			</or>
		</Trigger>
	</Question>

	<Question id="7" questionStatement="Random question?">
		<Answer text="Yes"></Answer>
		<Answer text="No"></Answer>
		<Trigger>
			<and>
				<question question="3" answer="1"></question>
				<not>
					<question question="4" answer="1"></question>
				</not>
			</and>
		</Trigger>
	</Question>

	<Question id="8" questionStatement="Question asked only if '21+' tag is
> 2">
		<Answer text="Yes"></Answer>
		<Answer text="No"></Answer>
		<Trigger>
			<tag name="21+" weight="2"></tag>
		</Trigger>
	 </Question>

	 <Question id="8" questionStatement="Question asked only if 'City bar
guide' asset is > 2">
		<Answer text="Yes"></Answer>
		<Answer text="No"></Answer>
		<Trigger>
			<asset name="City bar guide" weight="2"></asset>
		</Trigger>
	</Question>
</SubDialog>
```