The University of Southern Mississippi
The Aquila Digital Community

Dissertations

Spring 5-2014

Reducing Ambiguities in Customer Requirements Through Historical Rule-Based Knowledge in a Small Organization

Silvia Brum Preston
University of Southern Mississippi

Follow this and additional works at: https://aquila.usm.edu/dissertations

Part of the Computational Engineering Commons, and the Computer Engineering Commons

Recommended Citation
Preston, Silvia Brum, "Reducing Ambiguities in Customer Requirements Through Historical Rule-Based Knowledge in a Small Organization" (2014). *Dissertations*. 251.
https://aquila.usm.edu/dissertations/251

This Dissertation is brought to you for free and open access by The Aquila Digital Community. It has been accepted for inclusion in Dissertations by an authorized administrator of The Aquila Digital Community. For more information, please contact Joshua.Cromwell@usm.edu.

The University of Southern Mississippi

REDUCING AMBIGUITIES IN CUSTOMER REQUIREMENTS THROUGH HISTORICAL RULE-BASED KNOWLEDGE IN A SMALL ORGANIZATION

by

Silvia Brum Preston

Abstract of a Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

May 2014

ABSTRACT

REDUCING AMBIGUITIES IN CUSTOMER REQUIREMENTS THROUGH HISTORICAL RULE-BASED KNOWLEDGE IN A SMALL ORGANIZATION

by Silvia Brum Preston

May 2014

During the elicitation process the requirements for a software application are obtained from the customer. Customers often do not know how to clearly express the requirements of the application to be built, causing requirements to be ambiguous. Many studies have been found to cover different characteristics of the requirements elicitation process including methods for reducing ambiguities in requirements. The methods and findings of these studies were found to be too general when it comes to the specific domain of the requirements and knowledge about the requirements. In addition, some studies did not take into consideration the level of expertise of those users performing the process. The focus of this study is to reduce ambiguities in customer requirements for a specific domain through the use of a historical rule-based knowledge and a scripted process. Using a case study scenario, this study explores how ambiguities in customer requirements can be reduced using knowledge about specific requirements for Web-based forms. The scripted process is a step-by-step procedure utilized to guide a novice developer in reducing the ambiguities in customer requirements.

The proposed rule-based knowledge encompasses requirements of previously implemented Web-based applications.

The results of this study intend to improve domain knowledge sharing between novice and expert developers and domain experts while reducing ambiguities in customer requirements. The existence of ambiguities in requirements and the lack of knowledge about the domain, between customers and the development team, provide the context in this qualitative case study. The outcome of this study demonstrates how ambiguities in requirements can be reduced and easily understood by the development team while lessening the communication gap between all people involved. The impact of this study is relatively associated with the effort and time that goes into understanding requirements and reducing ambiguities.

COPYRIGHT BY

SILVIA BRUM PRESTON

MAY 2014

The University of Southern Mississippi

REDUCING AMBIGUITIES IN CUSTOMER REQUIREMENTS THROUGH HISTORICAL RULE-BASED KNOWLEDGE IN A SMALL ORGANIZATION

by

Silvia Brum Preston

A Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Approved:

Chaoyang Zhang
Director

Ray Seyfarth

Bikramjit Banerjee

Dia Ali

Tulio Sulbaran

Maureen A. Ryan
Dean of the Graduate School

May 2014

ACKNOWLEDGMENTS

First and foremost, I am grateful to God the Father Almighty for giving me the patience and strength I needed to carry on this study. Without his love, I would not have accomplished this work.

I would like to express my appreciation and to thank my advisor and committee chair, Dr. Joe Zhang, for giving me guidance and encouragement when I needed it the most. You were constantly pursuing me to complete this work and your enthusiasm was extremely helpful. Without his persistent help, this dissertation would not have been possible.

I would also like to thank my committee members Drs. Dia Ali, Tulio Sulbaran, Ray Seyfarth, and Bikramjit Banerjee for their support and assistance throughout this dissertation. Their collaboration was vital for the completion of this dissertation. Without their input and valuable feedback, this dissertation would have no significance.

My appreciation also goes to the users who tested this study. I don't know what I would do if it was not for their input during the case study of my dissertation. Thank you very much for all of your help. You know who you are.

I cannot forget to put into words my deepest admiration for the staff of the School of Computing. Especially, I would like to thank Crystal McCaffrey for her support and encouragement to conclude this dissertation. She was a great stimulus in my journey toward obtaining this degree.

Last but not least, I want to thank my husband, my family, and my in-laws for all the support, incentive, and love they have given me during this time. I love you all!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER I

INTRODUCTION

Background

Requirement Engineering (RE) is one of the most important disciplines in the development of software products. Successful and effective RE can improve risk management, quality, reusability, and productivity during the software development process. One of the main practices in RE is the elicitation process of software requirements. According to the Software Engineering Body of Knowledge (SWEBOK), software requirements can be defined as “a property which must be exhibited in order to solve some problem in the real world” (Committee, 2004). Requirements basically fall into two categories: 1) Functional requirements – describe the functions of the software i.e., what the software will actually do and; 2) Non-functional requirements – describe the constraints of the software or the quality requirements of the software. Software requirements are English like terms that describe the behavior of a desired object or entity and the functional aspects that are performed to modify the condition or the characteristics of an object. Requirements do not describe how a system is to be developed. They are mainly focused on the “what” and not on the “how”. Requirements main objective is to describe the needs and problems of the customer and not the solution or the development of the system.

The requirements that meet customer needs are often specified in the software requirements specifications. These specifications are derived from the requirements elicitation process. It is during the requirements elicitation process that customers describe and specify their needs to solve a problem. Customers often do not know how to express their needs of what they want implemented. It is a fact that during this early stage of the elicitation process that customer requirements are often malformed and not understood by the people involved in the process. Although customers understand their business, they are not always good in expressing what their business needs are. Many times the requirements produced fall short in quality, and in satisfying users' needs. Often customers do not have the knowledge to use existing methodologies for expressing requirements. The lack of knowledge causes poor and ambiguous requirements to be elicited. History has shown and it is a well-known fact that bad requirements lead to bad products.

When customers are not able to address the requirements needed for the software to be developed, developers and analysts can become beneficial in helping customers with this process. For example, a customer may suggest searching the database for a given student name. The developer knows searching a database may take a long time and that additional parameters are required. With a suggestion from the developer, the customer agrees to a change in their requirements. In order to understand customer needs and determine the requirements for the intended project, requirements analyst or a developer meets with the customer to elicit the requirements. The analyst's job is to ask the customer questions about the project and to examine the current behavior of the proposed project. Analysts may also suggest demonstrating similar projects in order to capture the requirements.

The focus of this dissertation is to provide a method for reducing ambiguities in customer requirements through the use of a collection of existing knowledge about specific requirements in a specific domain. The process of supplying similar requirements of existing projects as a method for capturing customer requirements can be beneficial in reducing ambiguities and valuable during reusability. The process can also reduce the communication gap among all people involved in the process by improving customer, analysts, and developers' communication. Often, it is the intensions or perceptions of each of these players that must be properly explored to determine the exact constraints of the system. For successful requirements engineering, it is important for the stakeholders to have a good bridge in communication. Each stakeholder has his/her own but very different perception of what is needed to build an effective product (Pfleeger & Atlee, 2006).

Statement of the Problem

For requirements to be of quality, it is necessary that the requirements be correct, complete, precise, consistent, verifiable, modifiable, and traceable (Toval, Nicolás, Moros, & García, 2002). Requirements that are not of good quality often cause problems during the software development process. Interpreting requirements correctly is a major problem in RE. Studies show that only about 42%-67% of requirements are delivered in a given project (Jacobs, 2007). Many industries cannot afford the consequences of not doing RE effectively and correctly, and ambiguous and inaccurate requirements can cost a company time, money, resources, and lost opportunities (Jacobs, 2007).

Requirements are often written in natural language even though notations, e.g., formal notations, diagrams, tables, patterns, and pseudo-code are available (Denger, Berry, & Kamsties, 2003). The process of eliciting software requirements involves different techniques that analysts and engineers use to collect the requirements. In his study, Coulin conducted and analyzed existing processes, methods, approaches and tools for eliciting requirements (Coulin, 2007). However, these techniques might not be intuitive to novice customers due to their lack of technical knowledge. Also, most methods for eliciting requirements do not support a scripted process for recording the activities in requirements elicitation and what needs to be done and by whom during the process. The overall process can also be affected when there is no analyst available and a novice developer is assigned to work with the customer. The elicitation process must be supported by a step-by-step procedure that fully describes the role of each person involved in the process and the steps for reducing ambiguities in requirements. The Software Engineering Institute (SEI) at Carnegie Mellon University addresses a scripted process, the step-by-step process for each area in the software life cycle (Humphrey, 2000, 2005). This cycle covers from the requirements, design, code, and test to acceptance. Although the process gives insights into requirement generation and a process for the overall software development life cycle, it lacks the details and provides no method to help in reducing ambiguities in customer requirements.

The objective of this dissertation is to explore, implement, and analyze a rule-based framework for reducing ambiguities in customer requirements during the elicitation process. The proposed framework aims to help the less experienced domain expert and novice developers to write functional requirements with fewer ambiguities. The framework incorporates a scripted process and a conceptual method to aid the users when obtaining requirements. The scripted process defines in details the steps for operating the conceptual method and supported materials for reducing ambiguities. The conceptual method incorporates a collection of similar requirements of previously implemented projects in a specific domain.

Research Questions

The overall goal of this study is to demonstrate that requirements can be improved through reducing ambiguities with the use of a rule-based framework while also improving the communication between novice customers, novice developers, and expert personnel. This is specifically accomplished when novice customers and personnel work together in the process of acquiring the requirements.

The proposed framework, as shown in Figure 1, supports an ontology representing the requirements for a specific domain and a knowledge-base containing requirements instances of existing application projects. How the framework is used and how to incorporate its results is described in the proposed scripted process.

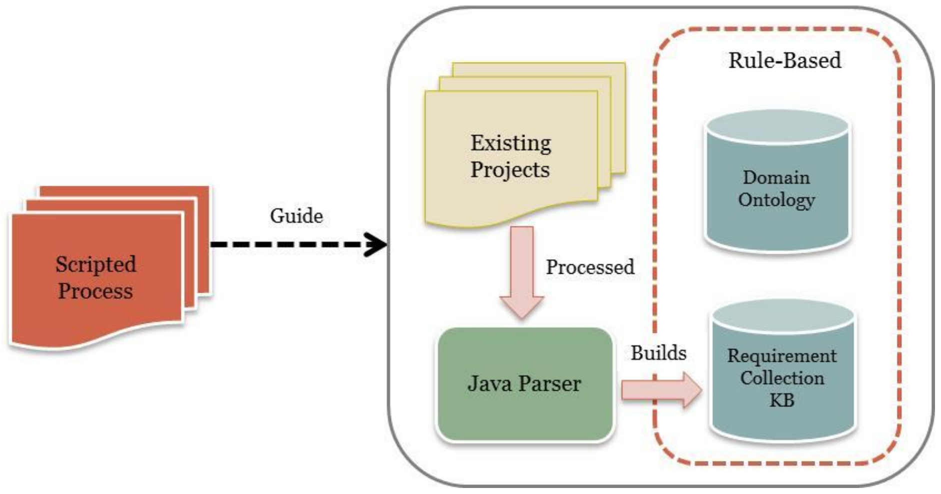


Figure 1. Proposed Framework. A framework supported by a step-by-step scripted process. Existing projects are parsed through a Java parser. A rule-based system using the Jess Rules language utilizes the parsed information for its requirement ontology and domain knowledge.

The technique proposed in this study supports both a scripted process and a conceptual method that supports ambiguities reduction in new customer requirements and the reusability of requirements while improving the communication and understanding of the people involved in the process. The use of an ontology provides specification of conceptualization of the specific domain of Web-based forms. The ontology developed in this study allows the modeling representation of the concepts, attributes, and relations among HTML form concepts and SQL table concepts. The ontology includes information about each concept and allows for reasoning rules to operate on the knowledge representation. During the requirements elicitation process, an inexperienced developer is able to use the ontology as a guide for reducing ambiguities in customer requirements before the formal requirements specifications can be stated. In order to accomplish these objectives, the following research questions were established:

Research question 1. How can ambiguities be reduced from customer requirements and converted to a clearer set of functional requirements that is understood by all stakeholders?

Research question 2. What can be done to reduce the cognitive distance between the following two groups: (1) the inexperienced and experienced developers and (2) the customers and developers when it comes to eliciting functional requirements?

Significance of the Study

Although many requirement elicitation methods are present in the literature, not all processes fit the specific needs of a customer. Methods are often used in conjunction with other methods to better describe customer needs. Customers are the people who often write the requirements of what they want built. The requirements written are specified in terms that might not always be understood by the developers. Requirement analysts are often the ones to represent the customer when writing requirements. When analysts are not present, the customer interacts directly with the developer. This interaction between customer and developer can become a problem if the developer and/or the customer are beginners in eliciting requirements.

In this dissertation, the proposed process aims to address the issues in customer requirements and the issues in the interaction between developers and between customer and developers. The conceptual model seeks to reduce ambiguities in customer requirements during the elicitation of requirements. The step-by-step scripted process strives for directing developers on how to utilize the conceptual model for reducing ambiguities in customer requirements. The significance of this study will be demonstrated through a case study, and the results of this study will have a direct impact on the structure of requirements for Web-based forms. The results of this study will also have an effect on the communication between all people involved in the process. The idea is to bridge the communication gap between all persons involved by providing knowledge about the domain. Both customer and developers will benefit from the results of the proposed method when eliciting requirements and when reducing ambiguities in those requirements. Customers and developers will become more knowledgeable about the domain under discussion as they apply the proposed concept and scripted process to new requirements.

Summary of Remaining Chapters

In Chapter I, the problem was introduced. Also introduced were statement of the problem, research questions, and significance of the study.

Chapter II provides a review of the current literature related to the study presented here. It mainly discusses two areas that motivated most of this work, requirements improvement and reuse and the use of ontology and domain knowledge for processing requirements.

Chapter III provides the details of the methodology utilized in this research. In this section, the details of each phase of the methodology and the proposed ontology are explained. The step-by-step process for aiding in the proposed conceptual model is also described in this section.

Chapter IV covers the creation of the rules for populating the knowledge-base. This section also gives details about the selected sample and the additional rules for processing new requirements.

In Chapter V, a case study is developed, and the test results are presented. This section provides evidence that customer requirements ambiguities can be reduced and better requirements can be produced through the use of a historical knowledge-base and a scripted process.

Chapter VI provides a summary of the contribution of this research. It also provides the limitations of this study and suggestions for future research.

Four appendixes are provided and contain detailed information that supports this research.

CHAPTER II

REVIEW OF RELATED LITERATURE

Requirements Elicitation

The process for obtaining the requirements for a projected system involves requirements to be retrieved and detailed in the requirements specification document. The retrieval process is an interactive process that involves customers, analysts, developers, and anyone else familiar with the system to be implemented. These are known as the stakeholders. Each stakeholder has a contribution in the process for capturing requirements for a new system. Once requirements are elicited, as shown in Figure 2, the requirements are analyzed, specified, validated, and finally detailed in the Software Requirements Specification (SRS) document. This document represents a contract between customers and developers with specifics about the system to be built.

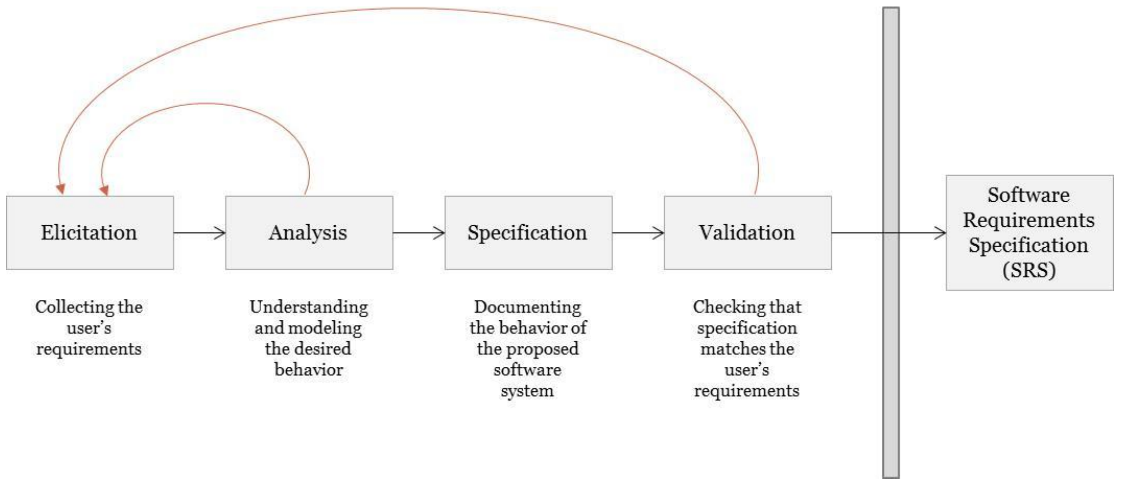

Elicitation
Analysis
Specification
Validation
Software Requirements Specification (SRS)
Collecting the user's requirements
Understanding and modeling the desired behavior
Documenting the behavior of the proposed software system
Checking that specification matches the user's requirements

Figure 2. Capturing Requirements for a Proposed System (Pfleeger & Atlee, 2006).

Collecting the user requirements is the main step in capturing requirements from all stakeholders involved. When requirements are not well understood, the analysis process takes place. It is in this process that requirements are analyzed and modeled. Ambiguities in requirements may require several meetings among developer, analysts, and customers in order to better comprehend the requirement. These meetings require another step in the elicitation process. When customer requirements are clear and well understood, customer requirements for the proposed system or application is documented. Each requirement is validated to make sure it meets a customer's needs before the final specification is fulfilled.

Different methods are used during the elicitation process to retrieve and document customer needs. The method selection affects the development of requirements due to the fact that a single method may not be appropriate for retrieving users' needs. A comparison of different techniques for requirements elicitation was elaborated. In the presented study, Zhang compared several methods for requirements development and recognized the "common factors that affect the method selection" (Zhang, 2007, p. 225). Zhang also discussed common guidelines for selecting a method for requirements elicitation "on which engineers can gain more experience on method selection in practice" (Zhang, 2007, p. 238).

Another study on existing processes, methods, and approaches on the state of the art of requirements elicitation was conducted. In his study, Coulin (Coulin, 2007) performed a paramount study on the different techniques in requirements elicitation. Using this study, Coulin proposed a tool and a procedure for requirements elicitation in a workshop with the collaboration of customers and analysts. The suggested approach takes into consideration novice users, and through a combination of processes and methods, users and analysts come together to elicit requirements. Though the proposed approach shows it can be implemented in a situational method, the approach lacks guidance on how to reduce ambiguous requirements once the workshop is completed and requirements are obtained. Also, the study does not give details on how the requirements can be stored and reused for the elicitation of requirements of future systems (Coulin, 2007).

During the requirements elicitation process customers, analysts, and developers perceptions must be taken into consideration. Often customers do not know how to interpret what they want without causing requirements to be misunderstood by the developers. In addition, multiple developers working on a single project have different perspectives of what the requirement entails. The impact of these subjects in requirements elicitation have been studied and evaluated in an experimental research (Arikoglu, 2011). Arikoglu (2011) concludes an experiment using two groups: users and "design actors" (p. 25). The study proposed uses scenario based design and persona approach to effectively evaluate the experiment. The experimental research is evaluated in order to understand the needs of the users and to guarantee there is understanding between the actors involved in the design of requirements (Arikoglu, 2011). The results of Arikoglu's investigation demonstrated that understanding users' needs is an important factor in requirements elicitation.

Requirements Improvement and Reuse

Currently in the literature there is a wealth of studies that focuses on the improvement of requirements specifications through a variety of methodology. It is known that requirements specification is the foundation for the whole software development process. It is essential that requirements be of quality and satisfy users' needs. For requirements of quality, it is necessary that the requirements be correct, complete, precise, consistent, verifiable, modifiable, and traceable (Toval et al., 2002).

Denger, Dörr, and Kamsties performed a survey on different studies that implemented methods and techniques in identifying problems in requirements (Denger, Dörr, & Kamsties, 2001). These studies provided guidelines on how to use natural language and sentence patterns processing for requirements written in natural language. The authors of this survey divided their focus into two categories. The first category describes specific language patterns for modeling requirements written in natural language (Lopez, Moreno, & Juristo, 2000; Ohnishi, 1994; Rolland & Proix, 1992). The second category characterizes the focus into linguistic rules and analytical keywords (Fabrini, Fusani, Gnesi, & Lami, 2000; Wilson, n.d.). Although these studies offer guidelines for improving and processing requirements written in natural language, there are some restrictions that need to be taken into consideration. For example, many of these studies offer no guidance in the correction of deficiencies found in requirements. In addition, these studies offer little to no support for the reusability of existing requirements.

In a more recent study (Kamalrudin, Hosking, & Grundy, 2011) on improving the quality of requirements, Essential Use Cases (EUCs) interaction patterns are used to link natural language requirements elements to each corresponding abstraction pattern. The tool provides a library of acceptable EUC patterns for matching against EUCs in order to determine if the use case model is correct, complete, and consistent. While this approach shows improvement in requirements written in natural language, the presented process does not fit in the work presented in this dissertation. The idea of using EUCs interaction patterns may be a suitable procedure for the projected set of requirements produced from the study employed in this dissertation.

When it comes to reusing requirements, different studies show methods for reusing requirements in different ways. In one study about reusability of software the authors described software reuse to be the only practical approach that can produce the productivity increase and the quality that the software industry needs (Mili, Mili, & Mili, 1995). The advantages of reusability are better when the abstraction level is raised and not only through requirement reusability, but also through designs and specifications reusability (Cybulsky & Reed, 2000). There are several approaches to requirements reusability, but the most successful method of requirements reusability should address the three major approaches: text processing, knowledge management and process improvement (Cybulsky & Reed, 2000).

One prominent way to address requirements knowledge reuse is to use pattern-based requirements (Franch, Palomares, Quer, Renault, & Tudor, 2010). As previous studies suggest, patterns can be employed to process requirements written in natural language during the analysis stage of software requirements. Barreto, Benitti, and Cezario (Benitti & da Silva, 2013) proposed a requirement reuse approach for eliciting and specifying requirements. The proposed approach utilized patterns catalogs for structuring knowledge for requirement writing while allowing traceability for the identification of new requirements from reused requirements. In the process, a pattern from the catalog is chosen for each *system requirement* and added to the requirements specification document. It has been suggested that without the use of a pattern, there is no reusability. The studies presented so far seem too general or too specific in scope and are particularly devoted to the requirements specified in the Software Requirements Specification (SRS).

This dissertation focuses on the actual customer requirement during the initial phase of requirements elicitation before the specification document is drawn.

Another way for reusing software takes into consideration the cognitive distance between all stakeholders. In Krueger (1992), the author produced a major survey of the software reuse literature where various approaches to software reuse was described. Krueger evaluated the effectiveness of reuse techniques in terms of cognitive distance. He determined the most effective technique in software reuse was automation of the abstractions in a reuse technique to an executable implementation (Krueger, 1992). According to him, for an efficient technique of software reuse there must be a common understanding "between the initial concept of a system and its final executable implementation" (Krueger, 1992, p. 136). This statement can also be applied to requirement elicitation and reuse. The efficiency in requirements elicitation and reuse is dependent on the common understanding between the initial process of eliciting the requirements and the implementation of the requirements specification document, which is also known as the SRS (Software Requirements Specifications).

Comparable to requirement reuse, other approaches encompass the use of methodologies for recycling requirements by analyzing and processing existing requirements of similar systems (Heumesser & Houdek, 2003; Knethen, Paech, Kiedaisch, & Houdek, 2002). One approach includes the construction of a tool for analysts to define requirements of similar systems (Kitazawa, Osada, Kamijo, & Kaiya, 2008). The tool in this study provides a list of requirements of existing systems allowing analysts to choose candidates of constraints in order to build a skeleton of requirements specification for a new system. While the tool provides a list of existing requirements to be chosen, there is no reasoning about the data. Analysts are assumed to be able to define requirements completely, correctly, and efficiently.

In another study related to reusability, Di Stefano and Menzies (2002) performed three machine learner's tests on a reusable data set (Di Stefano & Menzies, 2002). The goal in this study was to improve software reusability programs by using a combination of learning techniques. The data set was tested using the following learners: association rule, decision tree induction, classification rule, and treatment learners. The authors concluded that the major factor for success is "Human Factors" (Di Stefano & Menzies, 2002, p. 249). In addition, the authors found that multiple learners are necessary to identify necessary patterns in their data sets.

Evidently the reusability of requirements has an enormous impact on improving requirements in addition to leading to a better understanding of their details. The overall process of requirements involves a large amount of work by all parties involved from the elicitation of the constraints of the system all the way to producing the requirements specification document. The process of reusing requirements is beneficial to processing requirements which allows for the reuse of models, code, and other artifacts while reducing development time and improving the quality of the requirements (Benitti & da Silva, 2013).

Ontology Based Requirements

In the literature, there are studies that propose the use of ontology for the elicitation, analysis, specification and validation of requirements. The use of ontology has been especially useful during the requirements elicitation process. Domain ontologies are often built to represent knowledge about certain domains. In (Omoronyia, Sindre, &

Stålhane, 2010) the authors experiment the construction of a domain ontology for guiding users during requirements elicitation. Domain ontologies are built as per "existing technical standards which the specified requirements need to be compliant with" (Omoronyia et al., 2010, p. 189). The study presents an organized method for building domain ontology through text extraction in technical documents and the semantic process in the domain of transport. The method proposed by the authors improves the efficiency of building ontologies via technical documents, but experiments show effectiveness problems in addition to lacking techniques for reducing ambiguities in the proposed requirements.

The applicability of domain knowledge for requirements elicitation has also been studied. In (Kaiya & Saeki, 2006), requirements are elicited from requirements specifications written in natural language. The ontology built in this study represents a set of new requirements as concepts and relationships that are mapped through rules of inference. The technique proposed provides quality estimation for requirements, but the system lacks keyword matching, which could improve the meaning of requirements written in natural language.

A tool for converting requirements in UML model to ontology is described in Kroha, Janetzko, and Labra (2009). The proposed tool TESSI aids the analyst to write UML model for the requirements in addition to improving and reducing confusions in the requirements. The tool converts the UML model into the corresponding ontology model "that can be verified and compared with the domain ontology model to find contradictions" (Kroha et al., 2009, p. 34). The presented work and tool assert requirement specifications can be improved using ontologies by transforming the structural parts of UML models into ontologies to find “contradictions and inconsistencies in UML models” (Kroha et al., 2009, p. 36). Although the use of ontology has been presented to be useful in the area of supporting consistency in requirements specifications modeled in the UML model, the study proposed in this dissertation is concentrated on the initial set of functional requirements during the elicitation phase of RE.

The use of a knowledge-base allows for requirements reusability. It is a known fact that reusable requirements improve significantly the productivity and the quality of the final software product (Cybulsky & Reed, 2000). In one study, Zong-yong, Zhi-xue, Ying-ying, Yue, and Ying demonstrate the use of multiple ontologies as being essential in the elicitation and reusability of requirements (Zong-yong, Zhi-xue, Ying-ying, Yue, & Ying, 2007). The multiple ontology proposed includes a task ontology which combined with the domain knowledge helps obtain requirements that are relevant to the domain. These ontologies used together have the potential to allow requirements reuse. These approaches have so far been restricted by complicated frameworks that have limited scopes and the inability to coordinate and cooperate with other approaches.

In a different study, Dzung and Ohnishi (2009) discuss an ontology-based requirements checking tool (Dzung & Ohnishi, 2009). This tool maps initial requirements to functions in a domain ontology as input in a reasoning cycle. This cycle goes on until no new mandatory, redundant, or inconsistent requirement is found. Requirements sentences are parsed into verbs and nouns and then compared to a node in the ontology. Rules are used to reason about requirements using ontology, and if there is an error, the rules determine if the requirement should be added or not added to the list. Questions are generated to customers when one of the issues is found in the requirements. Although this is a good approach, the tool requires experienced users. It is assumed the user has experience in requirements elicitation. The authors provide no further details about the possibility of reusing the ontology. Also, the reasoning about requirements is based on new requirements. Historical requirements are not mentioned in the process. Finally, the questions generated to the customer are not specific as they relate to the data.

Another study in the area of ontology is proposed for describing business requirements and software attributes in terms of ontologies (Kluge, Hering, Belter, & Franczyk, 2008). In this study, ontologies are used in a semi-automated reasoning about the suitability of a certain software product. The approach proposed in this study does not provide algorithms to support the matching between the ontologies. The ontologies are built dynamically as new business requirements are specified. The authors profess that as of yet, no prior research has been done in the area of developing ontologies for existing software applications.

Most studies presented so far lack the presence of a guided process for the creation of an ontology. Another issue, is the lack of instructions about how to use the ontology to build the knowledge representation of the domain under discussion when defining requirements. Novice developers and customers often do not have the expertise of a requirements analyst to clearly define the requirements for a given application. At times, even analysts are in fact poorly trained or are not present in a limited budget organization. To address the problem of the absence of a guided process during requirements definition, Souag (2012) proposes a guided process for eliciting and defining requirements in the security domain. Once the requirements are elicited, the requirements are analyzed through the domain ontology for mapping and reasoning about the requirements. Although the study presented is focused on requirements for the domain of security, only a brief introduction is given on how the ontologies were built, and there are no results on the efficiency of the proposed work. The author suggests additional work is being explored to validate the results of the case.

In summary, the studies found in the related literature presented different approaches to requirements elicitation and processing. Each proposed work was short of one or more important factors characterized in this dissertation. The proposed work in this dissertation encompasses five characteristics: user experience, the definition of a static ontology for a specific domain, use a rule-based language for reasoning about knowledge to allow reuse of existing requirements, implement a step-by-step procedure for requirements elicitation process for both novice and expert domain users and analysts and finally, extend a historical knowledge-base for requirements through keyword matching. Table 1 summarizes some of the related work described in this section based on the characteristics of this dissertation.

Table 1

Related research and dissertation characteristics

Related Literature	User Experience	Ontology	Rules for Reasoning about Knowledge	Scripted Process and supported forms	Historical Knowledge Base
Kaiya and Saeki (2006)	Analysts do not have domain knowledge	Concepts and relationships domain ontology; lightweight semantic processing	Rules of inference for semantic processing	Procedure for improving and extending requirements; no related historical knowledge was presented	
Zong-yong et al. (2007)	Experienced developers and analysts	Multiple ontology definition for requirement processing, allow reusability of requirements	Scripted process exists for defining the ontologies and how to use the proposed ontologies during requirements elicitation; there are no rules defined and no related historical knowledge was presented		
Kluge et al. (2008)	Experienced business requirements analysts	Rudimentary matching between business requirements and software functionality ontologies	Semi-automated reasoning; no supported process was presented and no related historical knowledge		
Kitazawa et al. (2008)	Experienced analysts	Tool contains a mode for each step in the process; no ontology is proposed and no rules for reasoning about knowledge is presented			Tool contains functions of existing similar systems; Selection of common and related requirements of existing systems for new system

Table 1 (continued).

Related Literature	User Experience	Ontology	Rules for Reasoning about Knowledge	Scripted Process and supported forms	Historical Knowledge Base
Dzung and Ohnishi (2009)	User has experience in requirements elicitation	Ontology includes inheritance and aggregation relationships between verbs and nouns (semantic processing)	Reasoning about requirements is based on new requirements; no supported process was presented and no related historical knowledge		
Kroha and Labra (2009)	Experienced analysts	Ontology-based component for requirements specification; converts UML models into ontologies	Jess rules to check consistency; Pellet reasoner to check class hierarchy; no supported process was presented and no related historical knowledge		
Omoronyia, Sindre, & Stålhane (2010)	Domain experts to describe and document knowledge	Domain ontology based on technical documents; built using NL parsers	Rule-based approach using NLP techniques for capturing initial domain ontology from existing text; no available process and no historical knowledge		
A. Souag (2012)	Both novice and experienced analysts	Security ontology for processing textual security requirements and corresponding models	Rules for reasoning about knowledge of security requirements	Guided approach for supporting the development of requirements adapted to the definition of security requirements; no related historical knowledge was presented	

Table 1 (continued).

Related Literature	User Experience	Ontology	Rules for Reasoning about Knowledge	Scripted Process and supported forms	Historical Knowledge Base
This dissertation	Novice developer, inexperienced customer, no analyst available; limited budget organization	Ontology based on requirements definition for Web-based form; conceptualization of HTML form elements and SQL table definition	Jess rules for reasoning about knowledge, keyword matching, and syntax processing	Step-by-step scripted process with supported forms for aiding in processing requirements	Historical knowledge related to Web-based form requirements is proposed for improving requirement definition and for allowing unambiguous formation of requirement sentences

As presented in Table 1, each related work listed on the far left column lacks one or more characteristics presented in this dissertation as shown in the first row in bold. The last row in Table 1 summarizes the work presented in this dissertation based on each aspect named.

CHAPTER III

RESEARCH METHODOLOGY

This research purpose is to describe and explore the use of ontology and reasoning to create a historical knowledge-base of existing application requirements. In order to achieve the desired results of this study, there was a need to develop a research methodology. According to Paul Leedy and Jeanne Ormrod, the methodology implemented in this study falls in the “Qualitative Case Study” research design category (Leedy & Ormrod, 2009). In a case study research methodology, “a particular individual, program, or event is studied in depth for a defined period of time” (Leedy & Ormrod, 2009, p. 137). In the case of this research, the construction of an ontology and historical knowledge-base for reducing ambiguities in customer requirements and possible reusability were produced, and a scripted process was provided.

In a case study scenario, it is possible to apply qualitative content analysis as “a method of examination of data material” (Kohlbacher, 2006, p. 1). Kohlbacher explores and argues “that qualitative content analysis could prove to be a useful tool for analyzing data material in case study research” (Kohlbacher, 2006, p. 18). Mayring defines content analysis as “an approach of empirical, methodological controlled analysis of texts within their context of communication, following content analytical rules and step-by-step models, without rash quantification” (Mayring, 2000, p. 1). When applying qualitative content analysis to analyze the data in a case study, there are basic steps that must be completed, as summarized in Figure 3.

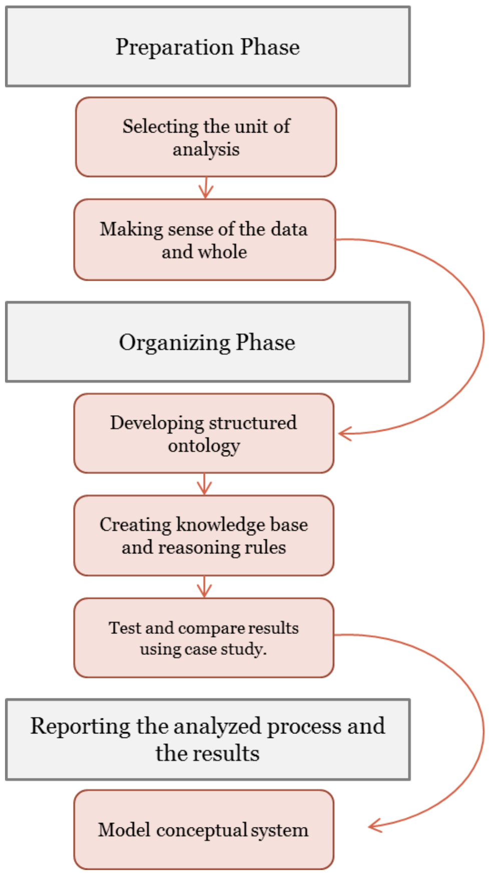

Preparation Phase
Selecting the unit of analysis
Making sense of the data and whole
Organizing Phase
Developing structured ontology
Creating knowledge base and reasoning rules
Test and compare results using case study.
Reporting the analyzed process and the results
Model conceptual system

Figure 3. Qualitative Content Analysis Process Phases. The phases of the process for the deductive approach include: preparation, organizing, and reporting (Mayring, 2008).

The process for analyzing content has two approaches: inductive approach and deductive approach. The inductive approach is recommended when the purpose of the study is new and there is not enough prior knowledge about the event being studied. In a deductive approach, the analysis is based on existing knowledge, and the focus of the study is on concept testing (Mayring, 2008). The methodology used in this dissertation meets the requirements of a deductive approach when it comes to applying content analysis methodology.

Phases

The focus of this dissertation is to analyze prior knowledge in requirements elicitation techniques and to test the concept of using ontology and a step-by-step procedure to reduce ambiguities in customer requirements. The methodology used in this research comprises of three phases summarized as follow:

Phase 1: Implementation of domain ontology and scripted process

Phase 2: Construction of knowledge-base

Phase 3: Testing of the proposed concept is conducted using a case study

Table 2 shows the steps in deductive analysis and the activities of this dissertation phases as they relate to the deductive analysis approach shown in Figure 3. Each step in the content analysis corresponds to an accomplished phase in this dissertation. This relationship between the steps and the phases of the presented study was necessary in order to achieve the goals of this dissertation. The details of each phase of this dissertation were established.

Table 2

Deductive Analysis Step (shown on the left side of the table) as they Relate to the Phases in this Dissertation (shown on the right side of the table)

Deductive Analysis Steps	Tasks
Preparation Phase Selecting the unit(s) of analysis Making sense of the data and whole (Who is involved? Where is this happening? When did it happen? What is happening? Why?)	Requirements Engineering, Ontology, Reasoning Rules, HTML forms, SQL tables Determine the people and environment involved in the study: customers and software developers Analyze the domain and its structure: HTML forms and SQL table concepts (Phase 1) Analyze and select a sample for the implementation of the knowledge-base (Phase 2)

Table 2 (continued).

Deductive Analysis Steps	Accomplished in this Dissertation
Organizing Phase Developing structured ontology Creating knowledge-base and reasoning rules Test and compare results using case study	HTML and SQL elements are parsed, structured and categorized to form the ontology; construction of the step-by-step procedure for handling conceptual system (Phase 1). Historical knowledge-base created in Jess rules using requirements of existing applications chosen for the sample (Phase 2). Test requirements ambiguities reduction (Phase 3).
Reporting the analyzing process and results Model conceptual system	Selection of a concrete case study for the conceptual system. Report results of reduced ambiguities in requirements through the use of a historical knowledge-base and the step-by-step procedure (Phase 3).

Methodology Details

The study conducted in this dissertation is focused in the area of requirements elicitation and analysis of the Requirement Engineering field. The implementation of a framework for reducing ambiguities in customer requirements encompasses an ontology, a knowledge-base, and a scripted process. The ontology comprises of classes representing Web-based form domain. The knowledge-base holds knowledge about instances of elements in a Web-based form. Each element in a Web-based form represents a requirement in the customer requirement list. The proposed scripted process aims in guiding the novice developers in operating the knowledge-base in eliciting and reducing ambiguities in requirements. The basic idea of the framework is to establish a practice that represents Web-based form requirements and the usability of these requirements through the practice of a scripted process. The supported structure allows for novice developers to process, analyze, and elicit requirements using a pool of knowledge about specific requirements for Web-based forms. Working together with the customer and making use of the scripted process, the novice developer, and the assistance of an expert developer, are able to inspect the suggested customer requirements and determine the requirements that are ambiguous and need refinement.

A summary of the methodological steps have been presented. The detailed description of each phase of the methodology is described.

Phase 1: Implementation of Domain Ontology and Scripted Process

The first step towards the investigation of a framework for reducing ambiguities in customer requirements is the process of learning about the knowledge domain and analyzing the data of the case being studied. From experience in the software development industry, it is a well-known fact that customer requirements have often been the target for ambiguities. The common ambiguity between customer requirements for different Web-based applications motivated the creation of an ontology to represent the knowledge about the domain for which the requirements represent. The knowledge domain under investigation encompasses customer requirements for the development of Web-based forms and applications. Research in the area of ontology was conducted, and no ontology has been found representing requirements for Web-based forms and database table structures as the study presented in this dissertation.

The motivation for creating an ontology is based on the fact that an ontology allows sharing of “common understanding of the structure of information among people and software agents” and “enables reuse of domain knowledge” (Noy & McGuiness, 2001, p. 1). In order to build the ontology, there was a need to understand the requirements. Requirements submitted to the software development unit being investigated are mostly for the creation of Web-based forms. Web-based forms are created using HTML tags and supported by a table structure or many tables in a database.

The ontology created establishes the foundation of domain knowledge for HTML tags and SQL table structures. The ontology was built in the Protégé ontology editor. Protégé is a knowledge modeling tool that allows for the creation of classes, slots, facets, and instances. The detailed description of Protégé is not the focus of this dissertation and can be found in a prior study (Noy, Fergerson, Musen, & Informatics, 2000). The domain ontology establishes the concepts of HTML and SQL tables and the relationships among these concepts.

The ontology contains properties and attributes of applications that contain only HTML items and also the properties and attributes of applications that contain both HTML items and SQL table items. An important part of Web-based applications includes the database in which an application uses to hold data entered in the form. As mentioned earlier, not all Web-based applications have a database for data storage. In this study, two types of Web-based applications are considered: 1) Applications that have a database backend, and 2) Applications that do not have a database backend.

SQL Class

SQLObjects. The tables that are part of applications are broken down in parts for requirement representation. Each column in a table represents a requirement and may or may not represent a field in a form. The following are the elements considered in a table for representing a requirement: *table name, column name, column data type, and column size*. The data type of a column represents by one of the following types: *varchar2, char, date, number, integer, decimal,* and *smallint*. As database tables of future applications are parsed, additional data types may be added to the knowledge-base. Figure 4 shows the *SQLObjects* class as it is related to SQL tables.

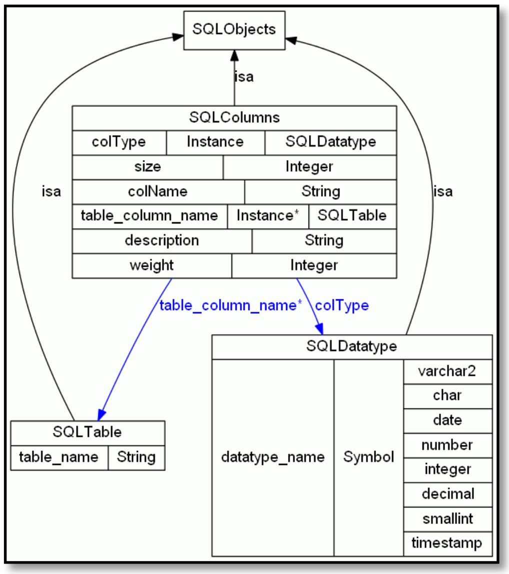

SQLObjects
isa
SQLColumns
colType | Instance | SQLDatatype
size | Integer
colName | String
table_column_name | Instance* | SQLTable
description | String
weight | Integer
isa
isa
table_column_name*
colType
SQLDatatype
datatype_name | Symbol | varchar2, char, date, number, integer, decimal, smallint, timestamp
SQLTable
table_name | String

Figure 4. SQLObjects Class. The class representing the SQL table structure.

The SQL table properties are represented by the *SQLObjects* class and its children, the *SQLColumns*, *SQLTable*, and *SQLDatatype* subclasses. These classes represent the structure of a table in a database. In this study, the focus was on Oracle and MySQL databases. A table in a database has a name and one or more columns. Each column in a table has a type and a size. Tables and columns in a database contain other properties that are beyond the scope of this study. Only elements that represent data of an application were considered.

Each class in the SQL table concept contains slots or fields and a type. Slots and fields are used interchangeably. The type of the slot was represented by the data type available in the ontology editor. In case of the Protégé ontology editor, the types available are: Any, Boolean, Class, Float, Instance, Integer, String, and Symbol. Due to the scope of this study, not all types are discussed.

SQLDatatype. The *SQLDatatype* class contains a single slot of type Symbol. In a frame based ontology, such as one created using the Protégé ontology editor, the type

Symbol refers to a list of constants a slot can have. In this case, the slot "datatype_name" can only have one of the following symbol constants: *varchar2*, *char, date*, *number*, *integer*, *decimal, smallint*, and *timestamp*. These constants values are based on the data type allowed when defining the columns of a table. Other data types are available, but these are the most used.

SQLTable. The *SQLTable* class contains one field. The "table_name" field was of type String and holds the name of the SQL table.

SQLColumns. The *SQLColumns* class contains several slots. Each slot represents the properties of a column in a SQL table. The slot "colType" represents the type of the column. It was an instance of the *SQLDatatype* class. The "size" slot is of type Integer and represents the size of the column. Not all types have a size and therefore, a default value of -1 was used. The slot "colName" is of type String and represents the name of a column. The "table_column_name" slot represents the name of the table. This slot is an instance of the *SQLTable* class and may contain one or more tables. If the table does not exist, this field is left blank. The "description" slot is of type String. It represents a description of what this column represents in an application. Finally, the "weight" slot is of type Integer and represents the weight of the column. The weight of the column is increased as often as it is chosen to be used in new applications.

HTML Class. Web-based applications contain HTML fields for data entry. Each field may or may not represent a field in a database table. Some applications store data in a database, and some retrieve the information entered via email.

For the HTML items, only items that are part of the form are relevant. This means, only those HTML items that are between the <form> and </form> tags of an

HTML page are considered. Furthermore, form elements that do not require user input are not to be considered. However, for the purpose of building the ontology, all elements in a form are included. These elements include the <label>, <button>, <fieldset>, and <legend> tags. The <label> tag is for defining a label for an input element with the <input /> tag. The <button> tag represents a button that can have text and image, but in a form it is preferred to use the <input> tag for buttons that require user input. The <fieldset> tag is for organizing similar elements in a form. The <legend> tag is for defining the caption for a fieldset element. As for the elements that are part of the form, the following items are considered: *<input />, <textarea>, <select>, <optgroup>*, and *<option>*.

HTMLObjects. The *HTMLObjects* class is another important class in the ontology. This class contains two subclasses, the *InputType* class and the *FormTagsType* class. The *InputType* class contains a single slot called "type", which is of type Symbol. The constant values of "type" slot are: button, checkbox, file, hidden, image, password, radio, reset, submit, and text. The *InputType* class was used as an instance type for a slot in the *Input* subclass of *FormTagsType*. The *FormTagsType* class contains three slots: "hasSQLObjects", "description", and "inApp", which are all of type String. These three slots are common properties of subclasses of the *FormTagsType* class.

The description of each subclass and corresponding slots, as shown in Figure 5, were derived from the HTML form tags definition as characterized on the w3schools website ("HTML Forms and Input," n.d.).

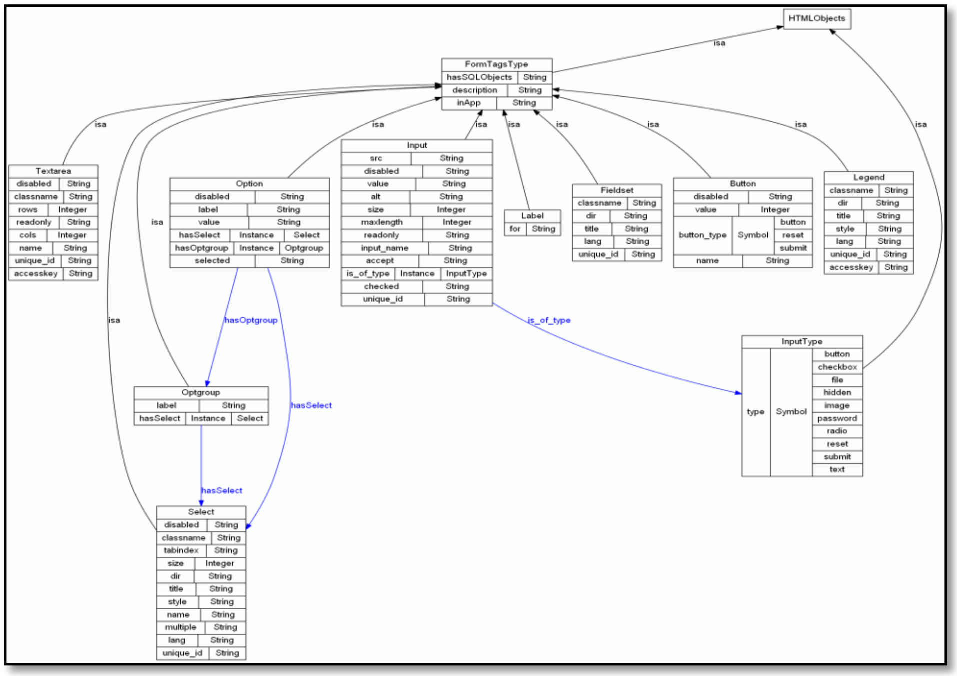


Figure 5. FormTagsType Concept Represents the HTML Form Tags Definition.

The subclasses of the *FormTagsType* class are described as follow.

Textarea. The *Textarea* class represents the textarea tag in a HTML form. In a HTML form this tag defines a multi-line text input control. This class contains 8 slots: “disabled” of type String, “classname” of type String, “rows” of type Integer, “readonly” of type String, “cols” of type Integer, “name” of type String, “unique_id” of type String, and “accesskey” of type String.

Select. The *Select* class represents the select tag in a HTML form. It is basically a dropdown list with options. This class contains 11 slots: “disabled” of type String, “classname” of type String, “tabindex” of type String, “size” of type Integer, “dir” of type String, “title” of type String, “style” of type String, “name” of type String, “multiple” of type String”, “lang” of type String, and “unique_id” of type String.

Optgroup. The *Optgroup* class represents an optgroup tag in a select field. When a select field has more than 10 items, it is recommended that related options in a select list be grouped together using the optgroup tag. This class contains 2 slots: “label” of type String, and “hasSelect”, an instance of the *Select* class. An *Optgroup* instance can only exist if there is a *Select* instance associated with it.

Option. The *Option* class represents the option tag in a select field in a HTML form. An instance of *Option* represents an option in a select list. This class contains 6 slots: “disabled” of type String, “label” of type String, “value” of type String, “hasSelect”, an instance of the *Select* class, “hasOptgroup”, an instance of the *Optgroup* class, and “selected” of type String. An *Option* instance must be part of a *Select* instance. An *Option* instance may or may not have an *Optgroup* object.

Input. The *Input* class represents the input tag in a HTML form. In a HTML form, the input filed can vary and be of different types. This class contains 12 slots: “src” of type String, “disabled” of type String, “value” of type String, “alt” of type String, “size” of type Integer, “maxlength” of type Integer, “readonly” of type String, “input_name” of type String, “accept” of type String, “is_of_type”, an instance of *InputType* class, “checked” of type String, and “unique_id” of type String.

Label. The *Label* class represents the label tag in a HTML form. It defines a label for an *Input* instance object. This class contains the “for” slot of type String.

Fieldset. The *Fieldset* class represents the fieldset tag in a HTML form. It is used to group related fields in a form by surrounding the fields with a border. This class has 5 slots in which all are of type String: “classname”, “dir”, “title”, “lang”, and “unique_id”.

Button. The *Button* class represents the button tag in a HTML form. This is just a push button on a form which can have text and image. The button created with the Input class is recommended for form processing. This class contains 4 slots: “disabled” of type String, “value” of type Integer, “button_type” of type Symbol, and “name” of type String. The “button_type” slot can only contain one of the following constant values: button, reset, and submit.

Legend. The *Legend* class represents the legend tag in a HTML form. An instance of this class is used as a caption of an instance of the *Fieldset* class. This class contains 7 slots in which all are of type String: “classname”, “dir”, “title”, “style”, “lang”, “unique_id”, and “access_key”.

Due to the scope of this study, the *Label*, *Fieldset*, *Button*, and *Legend* classes are not implemented in details in the case study scenarios. These classes are discussed here for future research purposes.

Complementary Classes

MapObjects. The *MapObjects* class is the mapping class which associates SQL objects and HTML objects that are part of an application. The conceptual graph, as shown in Figure 6 presents the relationship between the *MapObjects*, *Apps*, *FormTagsType* and *SQLColumns*.

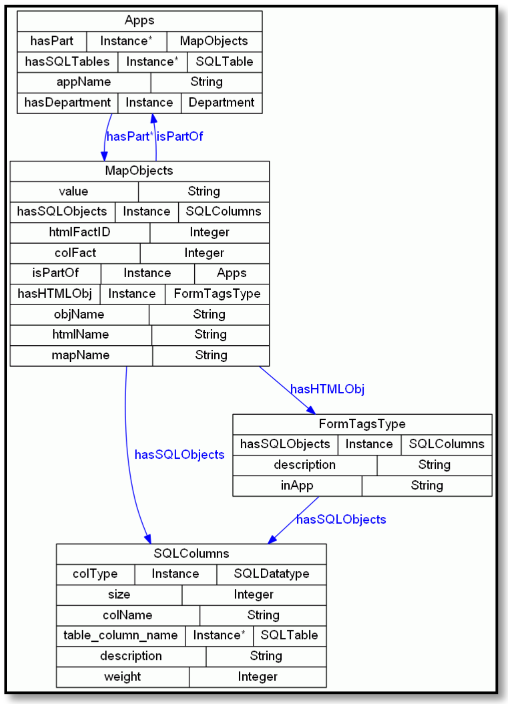

Apps
hasPart | Instance* | MapObjects
hasSQLTables | Instance* | SQLTable
appName | String
hasDepartment | Instance | Department
hasPart*
isPartOf
MapObjects
value | String
hasSQLObjects | Instance | SQLColumns
htmlFactID | Integer
colFact | Integer
isPartOf | Instance | Apps
hasHTMLObj | Instance | FormTagsType
objName | String
htmlName | String
mapName | String
hasHTMLObj
hasSQLObjects
FormTagsType
hasSQLObjects | Instance | SQLColumns
description | String
inApp | String
hasSQLObjects
SQLColumns
colType | Instance | SQLDatatype
size | Integer
colName | String
table_column_name | Instance* | SQLTable
description | String
weight | Integer

Figure 6. MapObjects, *Apps*, *FormTagsType*, and *SQLColumns* Concepts and their Relationships. Instances of *SQLColumns* objects are linked to *MapObjects* and *FormTagsType* instances.

The *MapObjects* class is applied to link HTML objects and SQL Objects. The definition of each slot and corresponding type of the *MapObjects* slots are shown in Table 3.

Table 3

Definition of Each Slot and Corresponding Type in MapObject Class

Slot	Type	Definition
value	String	
hasSQLObjects	Instance of SQLColumns	This object is a column
htmlFactID	Integer	The id of the corresponding HTML object
colFact	Integer	The id of the corresponding column object
isPartOf	Instance of Apps	The name of the application it is part of
hasHTMLObj	Instance of FormsTagsType	This object has a HTML element

Table 3 (continued).

Slot	Type	Definition
objName	String	This object's name. If this object has a column and a HTML object, then the name is the same. If this object has a column only, then the name will match the column name. If this object has only an HTML object, then the name will match the label of the field on the form.
htmlName	String	This is the same name as the name of the HTML object
mapName	String	If object does not have HTML, it represents the name of a SQL column, otherwise, it represents the unique ID of the HTML instance

Each slot in the *MapObject* class has a type, and it may or may not be associated to a SQL object it may or may not be associated to a HTML object. The slots "hasSQLObjects" and "colFact" contains the name of the corresponding SQL column and the fact ID for that column, respectively if the instance of this MapObject has an equivalent SQL column. The "htmlFactID" and "hasHTMLObj" slots have the corresponding HTML fact ID and HTML object type, respectively if this MapObject has an equivalent HTML instance. The "isPartOf" slot refers to the name of the application being defined. The "objName" slot contains the name of the requirement being defined. The "htmlName" slot refers to the name of the HTML object. The "mapName" slot represents the name of the corresponding column or the unique ID of the HTML instance.

Apps. The *Apps* class represents the applications that have been developed and each corresponding form object. As shown in Table 4, each slot in the *Apps* class is associated to another slot in another class in the ontology. The "hasPart" slot refers to the "isPartOf" slot in the *MapObject* class. The "hasSQLTables" slot refers the "table_name" slot in the *SQLTables* class. The "appName" slot contains the name of the application the requirements represent. Finally, the "hasDepartment" slot refers to the "deptName" slot in the *Department* class.

Table 4

Definition of Slots and Corresponding Type for the Apps Class

Slot	Type	Definition
hasPart	Instance of MapObjects	The MapObject object in this application
hasSQLTables	Instance of SQLTable	The SQL tables in this application
appname	String	The name/title of this application
hasDepartment	Instance of Department	The Department object this application belongs to

Department. The *Department* class contains only the "deptName" slot which is of type String. This is just a class to hold the different department names within the domain of discussion. Each department defined may have one or more applications. The relationship between the departments, applications, map objects, columns, and HTML objects are shown in the conceptual graph shown in Figure 7.

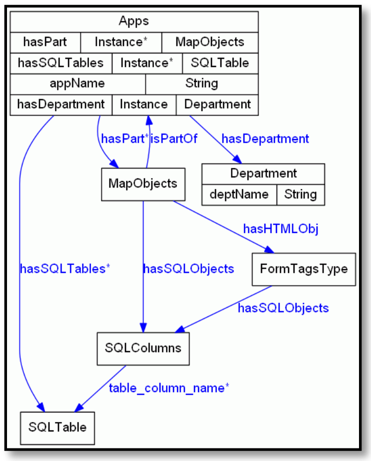


Figure 7. Department Class. The class department has a relationship between *Apps*, *MapObjects*, *FormTagsType*, *SQLColumns*, and *SQLTable* classes.

The relationship between all classes in the ontology is depicted in the conceptual graph show in Figure 8. Each class has a relationship with another class. Departments may have one or more Web-based forms. Each form contains elements that have HTML associations, and some elements may also be associated to a SQLobject.

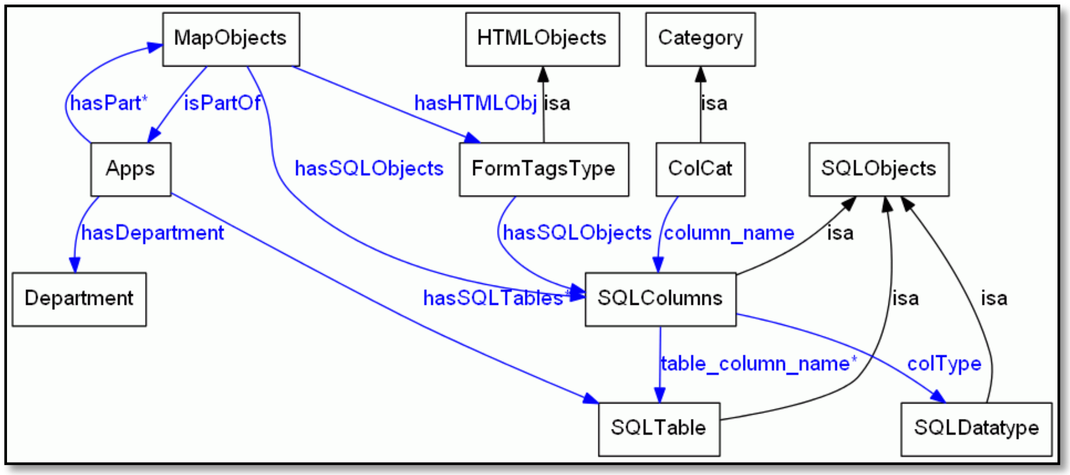
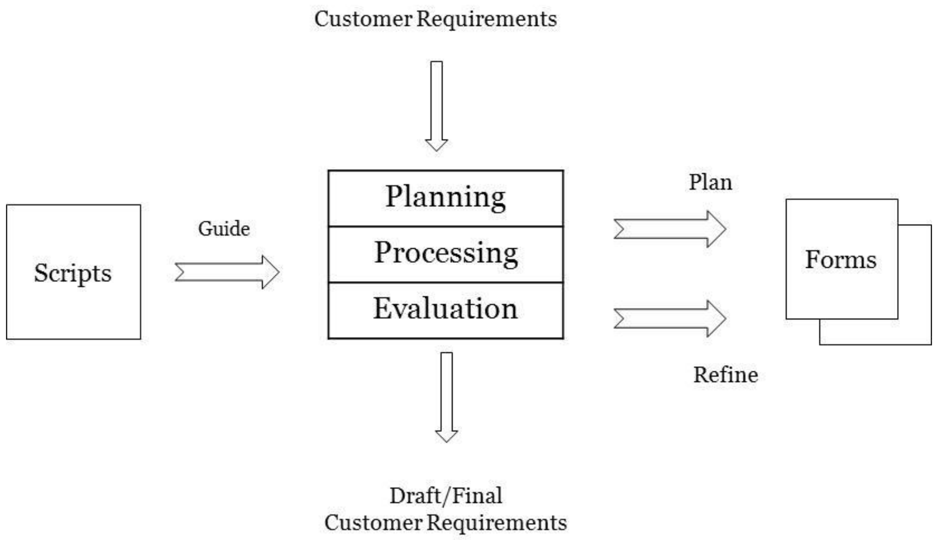


Figure 8. Relationship Between All Classes in the Ontology.

Once the ontology was built, reasoning rules and functions were implemented to support the relationships between HTML and SQL table concepts. Jess was the language chosen for reasoning about the concepts. Jess is a rule engine environment for the Java platform (Friedman-Hill, 2003). Jess is capable of reasoning data using knowledge supplied in the form of declarative rules. The reason behind using Jess is because it is a “small, light and one of the fastest rule engines available” (Friedman-Hill, n.d. para. 1). Jess is a powerful scripting language with full access to all Java’s APIs. Each class in the ontology is represented as a template in Jess. Assert statements allow for instances of templates to be created.

Scripted process. The implementation of an ontology and the rules for reasoning the relationship between the concepts led to the implementation of a step-by-step process.

This process is designed to aid in the use of the knowledge-base for reducing ambiguities in customer requirements. The process presented must be followed with the aid of organizational communication between two or more subjects. It is substantial that this process be applied during the beginning phases of requirements gathering. The process



suggested is divided into three separate stages as shown in Figure 9.

Figure 9. The Three Stages of the Proposed Process for Processing and Reducing Ambiguities in Requirements.

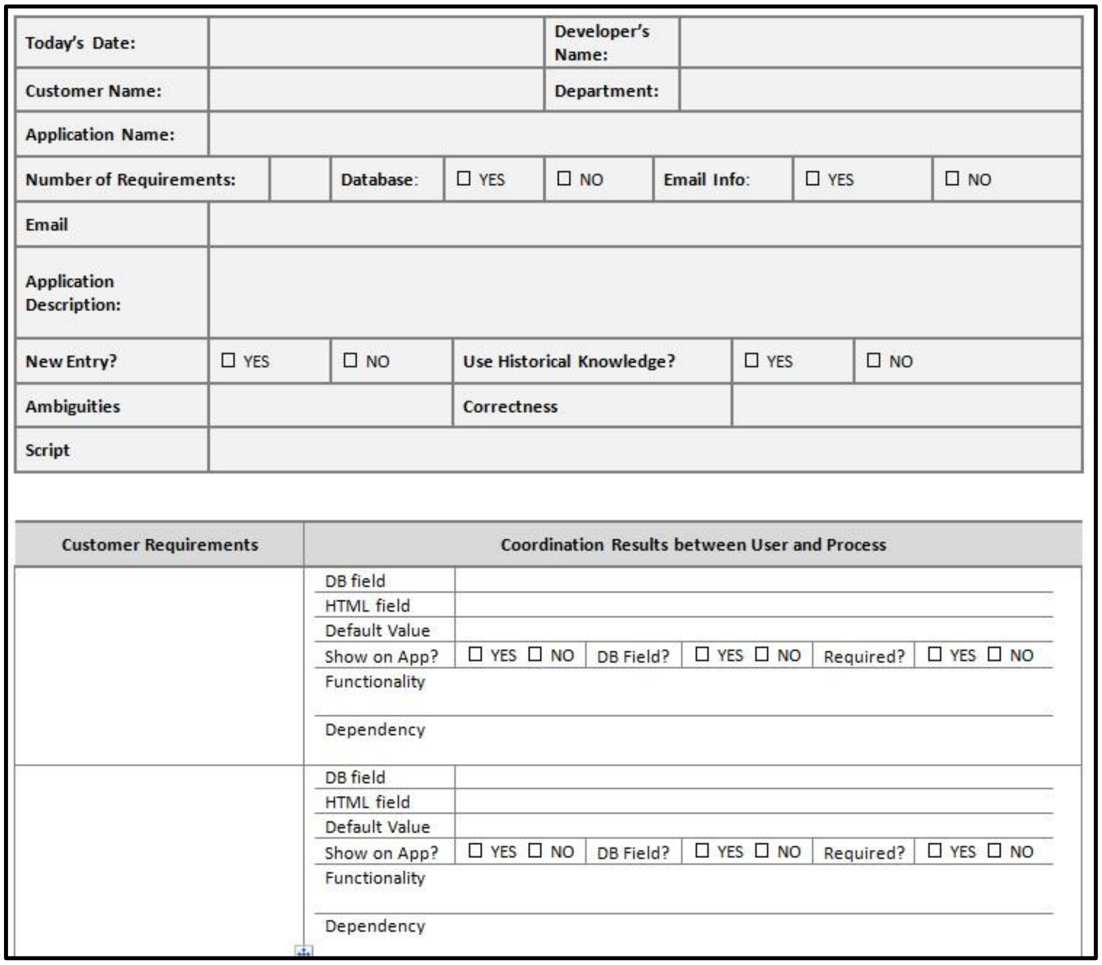
Prior to the planning phase, the entry criteria for following the process are identified. In this pre-planning phase, customer name, customer department, application name, and details are acknowledged. The planning, processing, and evaluation stages were derived from PSP (Personal Software Process), a well-known process in Software Engineering employed in software process improvement. Software engineers use PSP to track their performance during software development. The scripts associated to PSP allow engineers to log their time spent on each phase of software development and to make improvements in any stage of the process while consistently producing quality products (Humphrey, 2000).

In this dissertation, the planning, processing, and evaluation stages are also associated with scripts and a time recording log. These scripts are employed to guide novice developers in using the process to reduce ambiguities in requirements. The forms and instructions accompanying the three-stage process allow developers to record customer requirements, the results coordinated through the conceptual model upon processing each requirement, and the time spent processing the requirements. The scripts and associated forms and instructions can be found in Appendix A.

Planning. In this phase, the customer produces the initial requirements and stipulates the purpose of the Web-based form to be built. The customer here is assumed to have no prior knowledge in specifying requirements. Requirements are specified in one or more words in natural language, and no additional information is provided for each requirement. Novice developer enters the time spent in the Requirement Processing Time Recording Log form and input customer requirements into the Preliminary Customer Requirement form using the instructions provided with the form. The proposed log form was adapted from the Time Recording Log form provided in a previous study (Williams, 2000). In the Requirement Processing Time Recording Log form the developer will enter the time spent reviewing the set of requirements, time meeting with the customer, and time processing the requirements until a draft of the requirement is produced. The specifics about the new requirements for the new Web-based form to be created will be entered in the Preliminary Customer Requirement form. The functionality of each requirement is briefly covered in this study. Data entered in each field of the Web-based form can be saved to a database or it can be submitted to the customer’s email. There are security issues that may rise when private data is submitted via email.

This topic is beyond the scope of this study, but it must be considered when requirements are finalized. However, it is a good candidate as an extended part to this study.

The main form accompanying the three-stage process, as shown in Figure 10 allow novice developer to record customer requirements and the results coordinated through the use of the proposed process upon handling each requirement.



Today's Date:		Developer's Name:	
Customer Name:		Department:	
Application Name:			

Number of Requirements:		Database:	☐ YES	☐ NO	Email Info:	☐ YES	☐ NO

Email					
Application Description:					
New Entry?	☐ YES	☐ NO	Use Historical Knowledge?	☐ YES	☐ NO
Ambiguities			Correctness		
Script					

Customer Requirements	Coordination Results between User and Process					
	DB field					
	HTML field					
	Default Value					
	Show on App?	☐ YES ☐ NO	DB Field?	☐ YES ☐ NO	Required?	☐ YES ☐ NO
	Functionality					
	Dependency					
	DB field					
	HTML field					
	Default Value					
	Show on App?	☐ YES ☐ NO	DB Field?	☐ YES ☐ NO	Required?	☐ YES ☐ NO
	Functionality					
	Dependency					

Figure 10. Preliminary Customer Requirement Form.

The Preliminary Customer Requirement form is utilized in all three stages of the scripted process. In each phase of the process, a newer version of this form is applied in order to allow requirement changes to be recorded. Changes to each requirement are recorded as occurrence of the ambiguity factor. Each requirement the customer provides is listed in a separate row of the *Customer Requirements* column of the form. After handling each requirement through the process, the proposed result is recorded on the *Coordination Results between User and Process* column for each requirement. Each field in this column includes the suggested outcomes for the specific requirement. The functionality and dependency of each requirement is included here for textual matching purpose only. The actual functionality of a requirement in terms of how it is handled on the form is not covered in this study as it goes beyond of the original idea of this research. The fields in the shaded area of the Preliminary Customer Requirement form are filled out by both the customer and the developer at different phases of the process as follows. Instructions on how to complete this form can be found in Appendix A.

Processing. In this phase, a novice developer uses the proposed conceptual model to process each requirement in the Preliminary Customer Requirement form. If the novice developer cannot process a requirement, expert developer may become part of the processing phase if necessary. The proposed conceptual model suggests the correct way of writing the requirement based on existing knowledge about the requirement structure. Requirements for Web-based forms are required to be in a format that is understood by the development team. The proposed basic format of a requirement statement is shown in Figure 11.

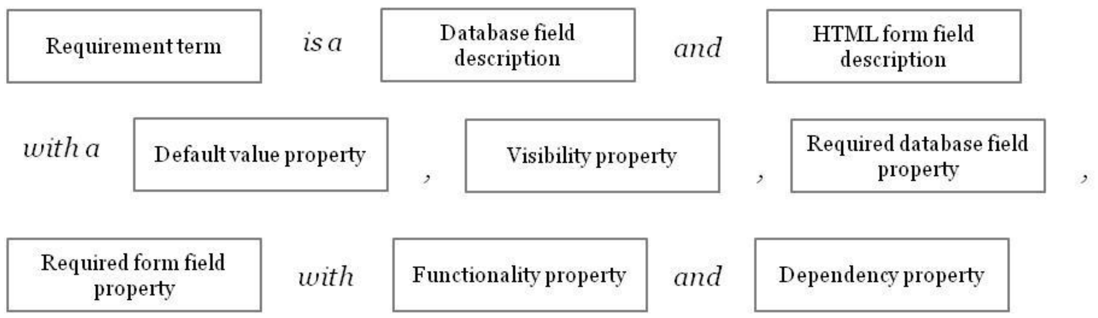
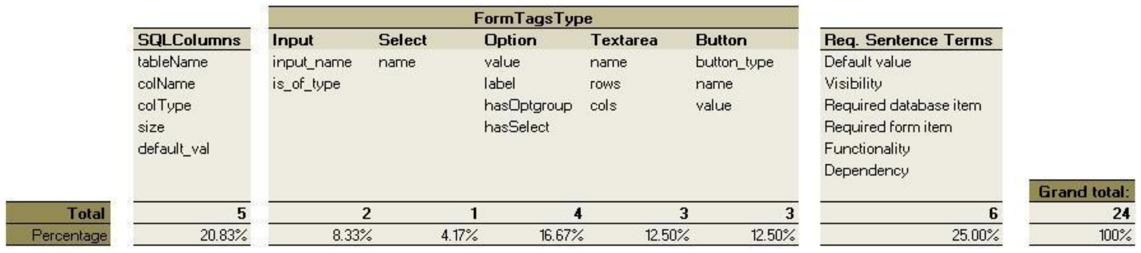


Figure 11. Requirement Sentence Format for Web-Based Forms.

The requirement term item is the actual customer requirement. The database field description and HTML form field description items are not dependent on each other. A requirement term may be a database field and exist in a HTML form. It can also be a HTML form item and not exist in a database field. Or it can be a database field and not a HTML field. The default value property describes the default value for this requirement in case there is no value entered in the form. The visibility property is concerned with the visibility of the requirement term on the form. The required database field property, and the required form field property items mean that the requirement is a required field on the form and must contain a value. These are only implemented when the requirement term is a database field and a form field or one or the other. The functionality property item describes the functional aspect of the requirement term. The dependency property is not always a required property. The format proposed here is the result of the coordination between the novice developer (also known as the user) and the conceptual model of the framework implemented in this dissertation.

When there is knowledge present, the conceptual model also allows for requirements to be matched against existing requirements. Using a collection of existing requirements is the ideal when proposing requirements for similar Web-based forms. Requirements that match to an existing requirement are added to the requirements specification draft document. Requirements that do not have a match or have one or more matches are considered ambiguous. A suggested description for an ambiguous requirement is produced using the same requirement sentence format seen in Figure 11. In the requirement sentence proposed, there are 24 terms that need to be taken into consideration when determining if a requirement is or is not ambiguous. Figure 12 summarizes these terms and the equivalent weight of each one in the sentence.



		FormTagsType						
	SQLColumns	Input	Select	Option	Textarea	Button	Req. Sentence Terms	
	tableName	input_name	name	value	name	button_type	Default value	
	colName	is_of_type		label	rows	name	Visibility	
	colType			hasOptgroup	cols	value	Required database item	
	size			hasSelect			Required form item	
	default_val						Functionality	
							Dependency	Grand total:
Total	5	2	1	4	3	3	6	24
Percentage	20.83%	8.33%	4.17%	16.67%	12.50%	12.50%	25.00%	100%

Figure 12. Terms in the Proposed Requirements Sentence.

As seen in Figure 12, there is a possibility of 24 terms in a requirement sentence. Each term is categorized based on the template it belongs. The actual requirement sentence terms are part of the requirement fact that is produced at the end of the process. The SQLColumns terms are part of the SQLColumns template. The FormTagsType terms are part of the FormTagsType template with the HTMLObjects template as the parent. The weight of each term is 4.17% and each group a total weight. The ambiguity characteristic of a requirement is calculated based on the number of terms in the sentence that are missing or incomplete. Therefore, in order to determine if a requirement is ambiguous or not the following formula must be used.

$$\textbf{\textit{Ambiguity}} = \frac{\textit{(Total \# of vague terms in requirement sentence > 0)}}{\textit{(Total \# of terms in a sentence)}}$$

$$\textbf{\textit{Unambiguity}} = \textit{(Total \# of vague terms in requirement sentence = 0)}$$

In the above definitions, the number of total terms in the produced requirement sentence that contain discrepancies must be greater than zero to be considered ambiguous. If each term is matched against an exact single term or each term is complete, the requirement is considered unambiguous. The collaboration between novice and expert developer determines which terms in the requirement sentence affect the ambiguity of the requirement sentence as a whole. After each requirement is processed, ambiguous requirements are analyzed and refined during the Evaluation step of the process.

Evaluation. During this phase, novice developer refines the ambiguous requirements through the proposed process. If assistance is necessary, the expert developer becomes part of this process. Each requirement term is corrected, completed, or changed as per the resulting meeting between the developers. A draft of all requirements is produced. With this draft at hand, novice developer and customer meet. In this meeting, the draft produced from the coordination between the developer and the conceptual model is analyzed. Customer analyzes each requirement in the draft to ensure the produced requirements meet the needs of the application to be developed. If any requirement in this draft does not meet customer needs, the processing phase is repeated and evaluation is carried out. This iteration is repeated until customer is satisfied with the list of requirements. Once the customer is satisfied with the list of requirements, a formal requirements specification document is elaborated.

The proposed three-stage process as shown in Figure 13 is incorporated into the conceptual model for customer and developers usage. The accompanying scripts aids in the manipulation of the conceptual model and forms usage. The forms permit customer to record requirements in order for the requirements to be refined and evaluated after the coordination between the conceptual model and the novice developer.

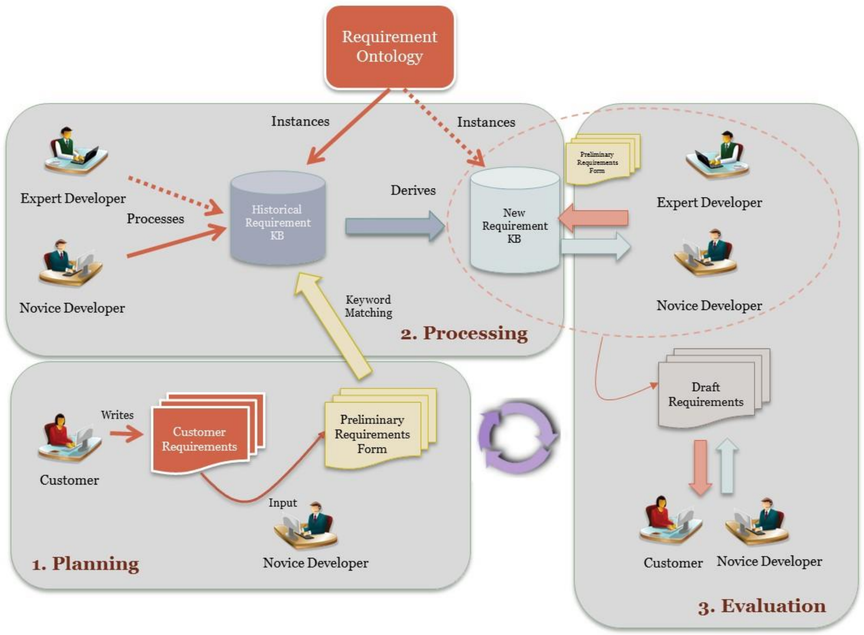


Figure 13. The Three-Stage Process as it is Applied to the Proposed Tool for Reducing Ambiguities in Customer Requirement.

Phase 2: Construction of the Knowledge Base

In order to build the knowledge-base, there is a need to select a particular sample of customer requirements. The purpose of the knowledge-base in this study is to provide a collection of requirements of previously implemented Web-based forms and applications. As discussed in Chapter IV of this dissertation, five previously implemented Web-based forms were gathered. The selection of existing applications was the key to the development of the requirements dictionary knowledge-base. A parser was developed in Java as part of this study to parse the HTML form tags and SQL create table script from which the requirements were derived. The ontology and knowledge-base were implemented using the Jess rules language. Instances of the ontology established the knowledge-base. Rules and functions were implemented to maintain the knowledge-base and to avoid the creation of inaccurate instances and to assure data in the knowledge-base is consistent. The results from parsing the HTML form tags and SQL create table script included assert statements. Assert statements were imported into the rule-based program in order to create instances of ontology concepts and to populate the knowledge-base.

Phase 3: Testing of the Proposed Concept using a Case Study

In this phase, testing of the proposed concept is conducted using a case study. The subjects selected for this study were software developers from a software development unit. The chosen subjects have different levels of experience. One developer is an expert in this area of requirements, and the second developer is a novice developer. They both work directly with customers and understand customer needs when it comes to requirements for Web-based applications. The case study scenarios, as described in Chapter V of this dissertation, include two sets of customer requirements written for two proposed Web-based form. The first set of requirements was processed in two ways: 1) no historical knowledge was present, and 2) historical knowledge of previously defined Web-based form requirements was present. The second set of requirements was processed only when historical knowledge was present. After each scenario, expert and novice developers met to review the draft and to discuss improvement in requirements. Statistical results from these scenarios were recorded and analyzed. The produced results were utilized to pinpoint the number of requirements that were ambiguous and unambiguous. These results of this study demonstrated whether or not the process was useful in reducing ambiguities in customer requirements.

CHAPTER IV

HISTORICAL RULE-BASED KNOWLEDGE

The implementation of an ontology described in Chapter III, lead to the development of a platform for loading data in the corresponding knowledge-base. The proposed platform includes a set of rules, functions, and queries for populating the knowledge-base with historical data of previously defined requirements and for processing new customer requirements. This chapter discusses the implementation of a historical rule-based knowledge for reducing ambiguities in customer requirements.

Knowledge Base Implementation

The knowledge-base was implemented using Jess, the rule engine program for Java. Jess was implemented using a plug-in in Eclipse IDE (Friedman-Hill, n.d.). In Jess, a set of templates, functions, queries, and rules were created. Templates in Jess are equivalent to classes in Protégé. The functions, queries, and the rules of the ontology were implemented in Jess in order to reason about the data. Jess provides several functions, but as with many programming languages, users can also define functions. Because Jess does not provide predefined rules in its language definition, rules were created. Also, a rule executes upon the existence of a fact that the rule refers to. In Jess, facts are instances of a template just as objects are instances of classes in Java. The following steps were taken to build the historical knowledge-base in Jess:

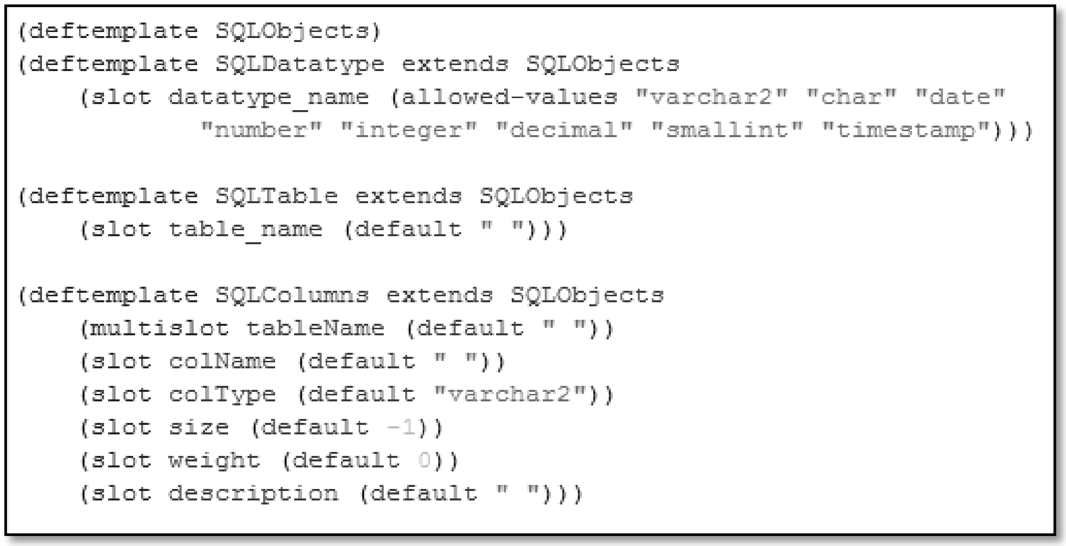
1. Install the Jess plugin in Eclipse
2. Defined Jess templates corresponding to each class in the ontology
3. Defined Jess rules, functions, and queries to reason about historical data

Installation of Jess Plugin in Eclipse

The Jess website (Friedman-Hill, n.d.) contains important information about how to install Jess as a plugin for Eclipse. Before downloading Jess, a form was filled, and contact was made with the person of contact for Jess. An email was received with instructions on how to obtain a free copy of Jess for research purpose only. The problem with this version was that it had an expiration date. A request was made to be able to use an unlimited version of Jess for students. Jess packages were download and added to Eclipse as plugins and features. Jess was also installed as a separate standalone platform. Once installation was complete, coding of the knowledge-base began.

Jess Templates Definition

The first step in processing the ontology was to create the Jess templates. Jess templates were created to represent each class in the ontology. Templates are like classes in Java. The name of the template corresponds to the name of the class. The slots of a template correspond to the properties of a class in Java. The name of a fact and its list of slots originate from its template just as an instance name and properties originate from a class. A template in Jess can extend one parent and inherits the parent's slots. Templates are created using the *deftemplate* construct. Figure 14 depicts the template and slot definition for *SQLObjects*, *SQLDatatype*, *SQLTable*, and *SQLColumns* classes.



```
(deftemplate SQLObjects)
(deftemplate SQLDatatype extends SQLObjects
    (slot datatype_name (allowed-values "varchar2" "char" "date"
            "number" "integer" "decimal" "smallint" "timestamp")))

(deftemplate SQLTable extends SQLObjects
    (slot table_name (default " ")))

(deftemplate SQLColumns extends SQLObjects
    (multislot tableName (default " "))
    (slot colName (default " "))
    (slot colType (default "varchar2"))
    (slot size (default -1))
    (slot weight (default 0))
    (slot description (default " ")))
```

Figure 14. Template Definition for *SQLObjects*, *SQLDatatype*, *SQLTable*, and *SQLColumns* Classes.

The *SQLObjects* class is the parent class for *SQLDatatype*, *SQLTable*, and *SQLColumns* classes. The “extends” keyword in the template definition of the *SQLDatatype* class identifies this class as being a child of the *SQLObjects* class. Detail of the template construct usage is beyond the focus of this dissertation and can be found in the Jess Rules manual (Friedman-Hill, n.d.).

Templates for each class defined in Jess are outlined. The Jess template for the *HTMLObjects* and its subclasses are listed in Table 5. Table 6 describes the *Department*, *Apps*, and *MapObjects* classes as Jess templates. Each template as shown may or may not have a parent template.

Table 5

HTMLObjects Class as Jess Template

Template Name	Slots	Properties	Parent Template
HTMLObjects			
InputType	Type	symbol type with allowed values of: button, checkbox, file, hidden, image, password, radio, reset, submit, text	HTMLObjects
FormTagsType	hasSQLObject	a SQLObject instance	HTMLObjects
	inApp	String	
	description	String	
Input	input_name	String	FormTagsType
	is_of_type	String	
	unique_id	String	
	value	String, default to an empty String	
		Int, default 1	
	size	Int, default 1	
	maxlength		
Select	name	String	FormTagsType
	unique_id	String	
Optgroup	label	String	FormTagsType
	has Select	String	
Option	value	String	FormTagsType
	label	String	
	selected	String	
	disabled	String	
	hasOptgroup	String	
	hasSelect	String	
Textarea	accesskey	String	FormTagsType
	classname	String	
	disabled	String	
	name	String	
	rows	String	
	cols	String	
	unique_id	String	
	readonly	String	
Button	button_type	Symbol with allowed values of: button, reset, submit	FormTagsType
	disabled	String	
	name	String	
	value	String	

Table 6

Department, Apps and MapObjects Classes as Templates

Template Name	Slots	Properties	Parent Template
Department	deptName	String	
Apps	hasDepartment	String	
	appName	String	
	hasSQLTables	String, multislot	
	hasPart	String, multislot	
MapObjects	objName	String	
	mapName	String	
	colFact	Int, default -1	
	hasSQLObjects	String	
	htmlFactID	Int, default -1	
	hasHTMLObj	String	
	htmlName	String	
	value	String	
	isPartOf	String, multi slot	

Rules, Functions, and Queries Definition

The set of rules defined in Jess allows instances to be created and processed. A rule contains 2 parts: the left-hand-side (LHS) and the right-hand-side (RHS). The LHS is matched against the corresponding facts in working memory. A collection of facts constitute the working memory in Jess. When an exact match occurs, the RHS of the rule is executed. Rules are executed when a fact is created, updated, and deleted. A fact is similar to an instance of an object in a programming language such as Java. Facts are created through assert statements. It is important to point out that rules are mostly created to keep the knowledge-base consistent and to avoid unwanted facts from being created. For instance, SQL table definition allows only for certain types of data to be defined. If a column is defined with a datatype that is not allowed, then a rule must exist to avoid the formation of such column. The rule created in this situation defines the column with a default datatype. Columns that have the same name and are in the same table also are not allowed to be instantiated. A rule was defined to retract such columns and to display an error message about the fault. A summary of certain rules defined in Jess for validating HTML and SQL instances in the knowledge-base can be found in Table 7.

Table 7

Dictionary Jess Rules for Validating HTML and SQL Instances

Criteria	Rule
The datatype for a column can only be one of the following: "varchar2" "char" "date" “number" "integer" "decimal" "smallint" "timestamp"	If a column is created with a datatype that is not allowed, the datatype for that column will be of a default type set as “varchar2” (defrule checkDatatype (declare (no-loop TRUE)) ?sqlC <- (SQLColumns (tableName ?tbl)(colName ?c)(colType ?t)(size ?s)(weight ?w)) (not (SQLDatatype (datatype_name ?t))) => (printout t "Type " ?t " is not a valid type. Changing to default \"varchar2\"." crlf) (modify ?sqlC (tableName ?tbl)(colName ?c)(colType "varchar2")(size ?s)(weight ?w)))
Do not allow columns with the same name and for the same table name to be created.	If a column is created for a table that already has a column with the same name, don’t allow the new column to be created. (defrule checkColInTable "Rule to make sure column fact doesn't already exist in the table" (declare (no-loop TRUE)) ?sqlc <- (SQLColumns(tableName ?table)(colName ?col)) (not (SQLColumns(tableName $? ?table $?)(colName ?col))) => (printout t "Column " ?col " already exist in table " ?table crlf) (retract ?sqlc))
When a column is created, make sure a table exist. If not, create the table.	If a column is created for a table that does not exist, display error message; delete new column created. (defrule checkTableExist (declare (no-loop TRUE)) ?sqlc <-(SQLColumns(tableName ?table)(colName ?col)) (not (SQLTable (table_name ?table))) => (printout t "Table name does not exist. Creating table." crlf) (assert (SQLTable (table_name ?table))))

Table 7 (continued).

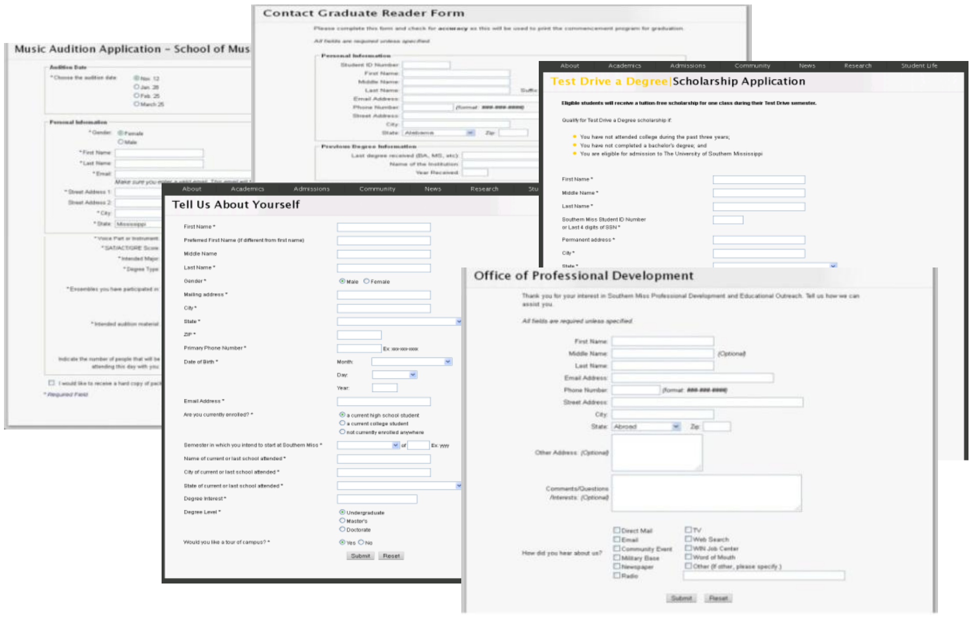
Criteria	Rule
When creating a form Input instance, make sure the type of the input field is a valid input type.	If an input field is created with a input type that does not exist, set the default input type to “text” (defrule checktype (declare (no-loop TRUE)) ?input <- (Input (input_name ?n)(is_of_type ?t)(size ?s)(unique_id ?id)) (not (InputType (type ?t))) => (printout t "Type " ?t " is not a valid type. Changing to default \"text\"." crlf) (modify ?input (input_name ?n)(is_of_type "text")(size ?s)(unique_id ?id)))
When creating a form optgroup label, make sure there exists a select instance associated with the optgroup. An optgroup can only be created if there is a select instance.	If an optgroup is created without an associated select instance, display error message and delete new optgroup created. (defrule checkOptgroupSelect (declare (no-loop TRUE)) ?optg <- (Optgroup(label ?label)(hasSelect ?select)) (not (Select(name ?select))) => (printout t "There is no valid Select instance for Optgroup " ?label crlf) (retract ?optg))
When creating an option, make sure there exists a select instance associated with this option. An option can only be created if there is a select instance.	If an option is created without an associated select instance, display error message and delete new option created. (defrule checkOptionSelect (declare (no-loop TRUE)) ?opt <- (Option(value ?value)(hasOptgroup ?optgroup)(hasSelect ?select)) (not (Select(name ?select))) => (printout t "There is no valid Select instances for Option " ?value crlf) (retract ?opt))

As shown in Table 7, the LHS of the rule is stated before the “=>” symbol while the RHS of the rule is stated after the “=>” symbol. In the “checkOptionSelect” rule, if there is a fact of the Option class that has the exact values for the “value”, “hasOptgroup”, and “hasSelect” slots, and it does not exist as a Select fact, the RHS of the rule is executed, which displays an error message and retract the Option fact. Several other rules were implemented to permit the formation of the knowledge-base. For instance, there was a need to create a rule to update the "colFact" slot in MapObjects class with the fact ID of the corresponding column. The fact ID of a fact can only be identified once the fact is created. The fact ID allows the relationships between the facts of different classes to be identified.

In addition to rules, Jess also allows for functions and queries to be implemented. Functions in Jess are executed when they are called to be executed. Unlike rules, functions do not depend on facts to be executed. Jess provides pre-defined functions and also allows user-defined functions. Jess functions and user-defined functions may or may not include parameters and may or may not be a return value function. Similar to rules, Jess provides queries. However, a query is invoked through a function call. A query has a left-hand-side, but it does not have the right-hand-side as in a rule. The results of a query include an object containing a list of all items matching the left-hand-side of the query. A number of functions and queries were implemented. For instance, a query was created to query all SQLColumns facts that match the "colName" slot to a given name. If one or match is found, the query returns all matching objects. This query is invoked from a function which process categories for columns and assert ColCat facts for the matching name.

Sample Selection

In order to populate the knowledge-base, there was a need to select a sample. The selected sample encompassed 5 Web-based forms implemented in a software development unit. These forms incorporated the necessary structure for both the HTML objects and SQL objects of the ontology. Instances of HTML objects and SQL objects were derived from these forms and loaded into the knowledge-base for historical use. The motivation for selecting the Web-based forms was due to the nature of this research which is focused on using historical knowledge-base to reduce ambiguities in customer requirements. In addition, the researcher had a significant contribution in the implementation of these 5 Web-based forms. Having an experience in building Web-based forms, the research found it necessary to improve the process for reducing ambiguities in customer requirements. The existing Web-based forms, as shown in Figure 15 constitute the data that was part of the knowledge-base.


Music Audition Application – School of Mus

Audition Date
*Choose the audition date [✓ Nov. 12] Jan. 28 Feb. 25 March 25

Personal Information
*Gender: [✓ Female] Male
*First Name:
*Last Name:
*Email:
Make sure you enter a valid email. This email will
*Street Address 1:
Street Address 2:
*City:
*State: Mississippi
*Voice Part or Instrument
*SAT/ACT/GRE Score
*Intended Major
*Degree Type
*Ensembles you have participated in
*Intended audition material
Indicate the number of people that will be attending this day with you
☐ I would like to receive a hard copy of pack
* Required Field

Contact Graduate Reader Form

Please complete this form and check for **accuracy** as this will be used to print the commencement program for graduation.

All fields are required unless specified

Personal Information
Student ID Number:
First Name:
Middle Name:
Last Name: Suffix
Email Address:
Phone Number: *(format: ###-###-####)*
Street Address:
City:
State: Alabama Zip:

Previous Degree Information
Last degree received (BA, MS, etc):
Name of the Institution:
Year Received:

About | Academics | Admissions | Community | News | Research | Student Life

Test Drive a Degree|Scholarship Application

Eligible students will receive a tuition-free scholarship for one class during their Test Drive semester.

Qualify for Test Drive a Degree scholarship if:

- You have not attended college during the past three years;
- You have not completed a bachelor's degree; and
- You are eligible for admission to The University of Southern Mississippi

First Name *
Middle Name *
Last Name *
Southern Miss Student ID Number or Last 4 digits of SSN *
Permanent address *
City *
State *

About | Academics | Admissions | Community | News | Research | Stu

Tell Us About Yourself

First Name *
Preferred First Name (if different from first name)
Middle Name
Last Name *
Gender * [✓ Male] Female
Mailing address *
City *
State *
ZIP *
Primary Phone Number * Ex: xxx-xxx-xxxx
Date of Birth * Month: Day: Year:
Email Address *
Are you currently enrolled? * [✓ a current high school student] a current college student not currently enrolled anywhere
Semester in which you intend to start at Southern Miss * of Ex: yyyy
Name of current or last school attended *
City of current or last school attended *
State of current or last school attended *
Degree Interest *
Degree Level * [✓ Undergraduate] Master's Doctorate
Would you like a tour of campus? * [✓ Yes] No

Office of Professional Development

Thank you for your interest in Southern Miss Professional Development and Educational Outreach. Tell us how we can assist you.

All fields are required unless specified.

First Name:
Middle Name: *(Optional)*
Last Name:
Email Address:
Phone Number: *(format: ###-###-####)*
Street Address:
City:
State: Abroad Zip:
Other Address: *(Optional)*
Comments/Questions /Interests: *(Optional)*
How did you hear about us? ☐ Direct Mail ☐ Email ☐ Community Event ☐ Military Base ☐ Newspaper ☐ Radio ☐ TV ☐ Web Search ☐ WIN Job Center ☐ Word of Mouth ☐ Other (If other, please specify.)

Figure 15. Selected Web-Based Forms to be Parsed and Incorporated into the Knowledge-Base.

The structure of these forms comprises of HTML form tags. These tags were extracted from the forms using a parse written in Java. The tags of interest here are the tags that are between the *<form>* and *</form>* HTML tags. The first step of the parser is to retrieve all content that is between the *<form>* and *</form>* tags. The content retrieved is stored in a list and then processed. The next step involves processing the items between the form tags. The items that are not related to a form field are ignored during the process. These tags include *<div>*, *<span>*, and any other tag related to the style of the form. Tags that are directly associated to the fields and their properties on the form are the ones included in this study. For instance, the HTML form snippet presented in Figure 16 shows the elements of a form. In the example shown, there is only one input field, which represents the First Name field on the form.

```
department = Gulf Coast Admissions
SQLTables = scholarships_form
<form name="form" id="form" method="post" action="index.php">
<input type="hidden" name="submitForm" value="form">
<div class = "formLayout">
<div class = "row">
<span class="leftColumn"><label for="firstname">First Name*
</label></span>
<span class="rightColumn"><input name="firstname" type="text"
id="firstname" size="30" maxlength="128" value=""></span>
</div></div></form>
```

Figure 16. Form Tag and Input Tag for Text Input Field.

The elements that were parsed in this snippet include: <label for="firstname">First Name, <input name ="firstname" type="text" id="firstname" size="30" maxlength="128" value="">.

The SQL create table structure for these forms were retrieved using SQL Developer (Oracle, n.d.). Figure 17 shows an example of a SQL create table script that was used as an input in the parser program. The SQL script for the create table, as shown, defines the name of the table, the columns in the table, and other properties related to the table. The significant items here are: the name of the table, the column name, the type of the column, and the size of the column.

```
--------------------------------------------------------
DDL for Table FORM
--------------------------------------------------------

CREATE TABLE "GCSCHOLARSHIP"."FORM"
(        "FORMID" NUMBER(10,0),
         "FIRSTNAME" VARCHAR2(128 BYTE),
         "MIDDLENAME" VARCHAR2(128 BYTE) DEFAULT '',
         "LASTNAME" VARCHAR2(128 BYTE),
         "EMPLID" VARCHAR2(11 BYTE),
         "STREET" VARCHAR2(128 BYTE),
         "CITY" VARCHAR2(32 BYTE),
         "STATE" VARCHAR2(2 BYTE),
         "ZIP" VARCHAR2(10 BYTE),
         "DOB" VARCHAR2(11 BYTE),
         "PRIMARYPHONE" VARCHAR2(12 BYTE) DEFAULT ' ',
         "EMAIL" VARCHAR2(128 BYTE),
         "LASTSCHOOL" VARCHAR2(128 BYTE),
         "LASTDATEATTENDANCE" VARCHAR2(24 BYTE),
         "SCHOLARSHIPSEMESTERYEAR" VARCHAR2(24 BYTE) DEFAULT ' ',
         "ALREADYAPPLIED" CHAR(1 BYTE) DEFAULT 'N',
         "DATEAPPLIED" VARCHAR2(24 BYTE) DEFAULT ' ',
         "APPLICATIONDATE" TIMESTAMP (6),
         "IPADDRESS" VARCHAR2(40 BYTE),
         "EXTRACTDATE" TIMESTAMP (6)
   ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS" ;
```

Figure 17. SQL Create Table. The SQL create table is a script utilized in the creation of database tables where columns are associated to elements in a form.

Parsing

In order to retrieve each individual field in the HTML form and the SQL objects corresponding to each field in the form, the forms and the SQL create table script had to be parsed. A Java parser was developed to parse the HTML form tags and the SQL create table script. Figure 18 illustrates the flow of the parsing process. The results of the parser program consist of Jess assert statements for the creation of Jess facts in the knowledge-base.

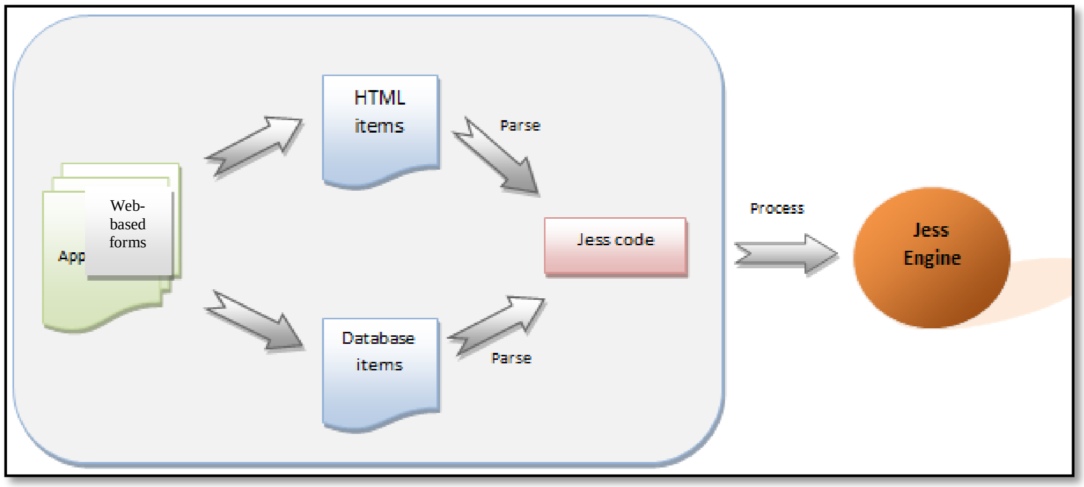


Figure 18. Web-Based Forms and Corresponding HTML and Database Items. Items are parsed into Jess assert statements and processed via Jess engine.

As indicated in Figure 18, the select forms were broken down into the HTML items and the database items. The HTML items represent the HTML form tags, and the database items represent the items in the SQL create table script. The parser program reads the HTML code and strips out all code that is not between the <*form*> and <*form/*> tags and the Jess code, for the assert statements are written to an external data file. The same was done with the SQL create table script. The name of the table, the columns, and the column properties were parsed, and the assert statement equivalent to these elements was written to the external data file. The program was executed two different times to produce two different files. Figure 19 shows examples of assert statements for the SQLTable and SQLColumns templates after the parsing of the SQL create table script:

```
(assert (SQLTable (table_name "GCScholarship_Form")))
(assert (SQLColumns (colName "firstname") (size 128)
        (colType "varchar2") (tableName "GCScholarship_Form")))
```

Figure 19. Assert Statements for Creating Facts of SQLTable and SQLColumns Templates. Facts are also known as instances, which are used to build knowledge.

In the example shown in Figure 19, facts of each template are created after execution of the assert statements in the Jess program. Values are given to each slot of the template. If a slot does not have a value and it has a default value in the template definition, the value of that slot will have the default value defined in the template definition. Otherwise, the slot will have a value of null.

The values given to the slots of each template are the values parsed from the existing Web-based forms. For the SQLTable template, the "table_name" slot contains the name of the table as defined in the SQL *create table* script. For the SQLColumns template, the "colName" refers to the name of the column; the "size" and "colType" slots refer to the size and datatype defined for this column. The "tableName" slot contains the name of the table defined in "table_name" slot of the SQLTable. Assert statements of SQLDatatype template are defined for all datatype values allowed in the Oracle SQL database (Lorentz, 2005). These assert statements are executed prior to any other assert statement. In addition to creating facts of each database item through assert statements, if the column is also a field on the form, the corresponding HTML assert statement is also created. Using the form code shown in Figure 16, assert statements were created for the Department, Apps and Input templates. Additionally, an assert statement for the MapObjects fact is defined for each item in the knowledge-base to show relationship. For instance, the "First Name" field is a SQLColumn fact and it is also an input field in the form, so the MapObjects fact is created to connect the two facts. Figure 20 illustrates the assert statements for Department, Apps, Input, and MapObjects for the "First Name" field.

```
(assert (Department (deptName "Gulf Coast Admissions")))
(assert (Apps (hasDepartment "Gulf Coast Admissions")
        (appName "scholarships_form")(hasSQLTables "GCSCHOLARSHIPS_FORM")))
(assert (Input(inApp "scholarships_form") (unique_id "firstname")
        (input_name "firstname")(maxlength 128) (size 30) (is_of_type
"text")
        (value " ") (hasSQLObjects "firstname")(description "First Name")))
(assert (MapObjects (objName "First Name") (hasHTMLObj "Input")
        (hasSQLObjects "firstname")(isPartOf "scholarships_form")))
```

Figure 20. Assert Statements for Department, Apps, Input, and MapObjects Templates. Upon execution of the assert statements, the corresponding rules are executed and specifics slots are updated.

Code Execution

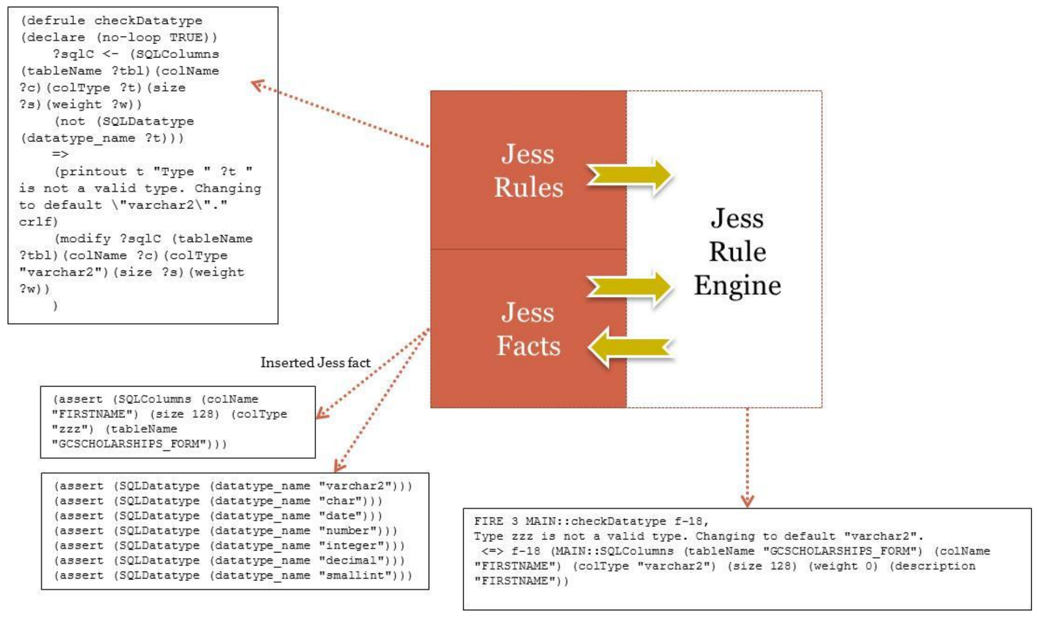
Once all templates, rules, functions, queries, and assert statements were created, the Jess program to populate the knowledge-base was executed. The assert statements for all the elements in the selected Web-based forms were defined as functions in separate files and loaded into the program. A main function was defined to control the flow of the program. Before executing the function with the assert statements for the elements on the forms, facts of SQLDatatype, InputType, and Category were created. The execution of the assert statements for these three templates was done prior to executing the assert statements for the elements in a form. This was a necessary step to avoid unwanted data to be instantiated.

The execution of certain rules is dependent on the existence of facts of templates. The assert statements created for the SQLDatatype template contained the allowed values as defined in the template’s slot. The allowed values for the datatype of a column in SQL were derived from the Oracle SQL database manual (Lorentz, 2005). This manual defines the allowed datatype values when defining columns in a SQL create table script. The assert statements created for the InputType templates included the allowed types of the

<input> tag in HTML. As mentioned earlier in this dissertation, the HTML input form tag can only be of type: *button*, *checkbox*, *file*, *hidden*, *image*, *password*, *radio*, *reset*, *submit,* and *text*. The assert statements for the Category template include the categories for the items in a form. Category facts are useful when classifying form fields that can be grouped together. For instance, the "First Name" field along with the "Middle Name" and "Last Name" fields can be part of the Names category. This is a convenient feature when there is a need for matching facts in the Names category. All facts in the Names category may be suggested for use. Note that no rules were executed after these assert statements were executed. The reason for this is that these are predefined constant values for each one of the templates; and therefore, there is no need for changing these created facts.

After the execution of the assert statements for instantiating facts of SQLDatatype, InputType and Category, the files containing the assert statements for the elements on the forms were loaded. The first file loaded was the file containing the SQLObjects assert statements. Assert statements for each table (defined as SQLTable) and the corresponding SQLColumns assert statements for each column defined in the table were defined and stored in the file. The second file loaded into the program contained the assert statement for HTML items. The statements in this file included the assert statements for Department, Apps, Input, Select, Option, Optgroup, Textarea, and MapObjects. As discussed earlier, these were the main HTML form elements considered in this study.

After the execution of the assert statements shown in Figure 19, rules for some facts are fired. Assert statements cause Jess facts to be created in working memory and rules to be fired. An example of a rule being fired upon the creation of a fact is seen in Figure 21.



```
(defrule checkDatatype
(declare (no-loop TRUE))
    ?sqlC <- (SQLColumns
(tableName ?tbl)(colName
?c)(colType ?t)(size
?s)(weight ?w))
    (not (SQLDatatype
(datatype_name ?t)))
    =>
    (printout t "Type " ?t "
is not a valid type. Changing
to default \"varchar2\"."
crlf)
    (modify ?sqlC (tableName
?tbl)(colName ?c)(colType
"varchar2")(size ?s)(weight
?w))
    )
```

```
(assert (SQLColumns (colName
"FIRSTNAME") (size 128) (colType
"zzz") (tableName
"GCSCHOLARSHIPS_FORM")))
```

```
(assert (SQLDatatype (datatype_name "varchar2")))
(assert (SQLDatatype (datatype_name "char")))
(assert (SQLDatatype (datatype_name "date")))
(assert (SQLDatatype (datatype_name "number")))
(assert (SQLDatatype (datatype_name "integer")))
(assert (SQLDatatype (datatype_name "decimal")))
(assert (SQLDatatype (datatype_name "smallint")))
```

```
FIRE 3 MAIN::checkDatatype f-18,
Type zzz is not a valid type. Changing to default "varchar2".
 <=> f-18 (MAIN::SQLColumns (tableName "GCSCHOLARSHIPS_FORM") (colName
"FIRSTNAME") (colType "varchar2") (size 128) (weight 0) (description
"FIRSTNAME"))
```

Figure 21. Jess Rules are fired upon Jess Facts in the Jess Rule Engine.

When the assert statements are executed, Jess facts are inserted into the working memory of the engine. In Figure 21, assert statements of SQLDatatype and SQLColumns are executed. After the execution of the SQLColumns assert statement, the “checkDatatype” rule is activated and fired. This rule displays an error message and modifies the fact that meets the criteria on the left-hand side of the rule. In this case, as summarized in Table 7, in the “checkDatatype” rule if a column is created with a datatype that is not an allowed SQLDatatype, the datatype for that column is changed to be of a default type, which is the “varchar2” datatype.

Another rule being fired and also summarized in Table 7 occurs when the SQLColumns assert statement is executed. When this happens, the rule "checkColInTable" is activated. This rule checks working memory for the existence of a SQLColumns fact with the same name as the "tableName" slot in the SQLColumns fact. If a column already exists with the same table name, the rule is fired, and this new fact is retracted. Otherwise, the rule is not fired, and the rule is deactivated. The rule becomes active again when a change occurs in the fact that matches the left-hand side of the rule or when a new fact is created.

Besides the rules summarized in Table 7, other rules in the program were defined. The execution of the assert statements in Figure 19 and 20 also causes additional rules to be activated and fired as shown in Figures 22a and 22b. The output of the program, as seen in Figures 22a and 22b show the rules that are activated and fired when the assert statements are executed.

```
==> f-1 (MAIN::SQLDatatype (datatype_name "varchar2"))
 ==> f-2 (MAIN::SQLDatatype (datatype_name "char"))
 ==> f-3 (MAIN::SQLDatatype (datatype_name "date"))
 ==> f-4 (MAIN::SQLDatatype (datatype_name "number"))
 ==> f-5 (MAIN::SQLDatatype (datatype_name "integer"))
 ==> f-6 (MAIN::SQLDatatype (datatype_name "decimal"))
 ==> f-7 (MAIN::SQLDatatype (datatype_name "smallint"))
 ==> f-8 (MAIN::InputType (type "checkbox"))
 ==> f-9 (MAIN::InputType (type "radio"))
 ==> f-10 (MAIN::InputType (type "reset"))
 ==> f-11 (MAIN::InputType (type "submit"))
 ==> f-12 (MAIN::InputType (type "text"))
 ==> f-13 (MAIN::Category (catName "Names") (subName " "))
 ==> f-14 (MAIN::Category (catName "Addresses") (subName "Home"))
 ==> f-15 (MAIN::Category (catName "Addresses") (subName "Mailing"))
 ==> f-16 (MAIN::Category (catName "Addresses") (subName "Degree"))
 ==> f-17 (MAIN::SQLTable (table_name "GCSCHOLARSHIPS_FORM"))
 ==> f-18 (MAIN::SQLColumns (tableName "GCSCHOLARSHIPS_FORM") (colName
"FIRSTNAME") (colType "zzz") (size 128) (weight 0) (description " "))
==> Activation: MAIN::checkDatatype :  f-18,
==> Activation: MAIN::checkColInTable :  f-18,
<== Activation: MAIN::checkColInTable :  f-18,
```

Figure 22a. Output of the Execution of Assert Statements for SQLDatatype, InputType, Category, SQLTable, and SQLColumns.

Figure 22a shows the output of the beginning of the execution of the program when assert statements for SQLDatatype, InputType, Category, SQLTable, and SQLColumns are executed. The Figure also shows the activation of “checkDatype” and “checkColInTable” rules.

```
==> f-19 (MAIN::Department (deptName "Gulf Coast Admissions"))
 ==> f-20 (MAIN::Apps (hasDepartment "Gulf Coast Admissions") (appName
"scholarships_form") (hasSQLTables "GCSCHOLARSHIPS_FORM") (hasPart " "))
==> Activation: MAIN::updateApps :  f-20
 ==> f-21 (MAIN::Input (hasSQLObjects "firstname") (inApp "scholarships_form")
(description "First Name") (input_name "firstname") (is_of_type "text") (unique_id
"firstname") (value " ") (size 30) (maxlength 128))
 ==> f-22 (MAIN::MapObjects (objName "First Name") (mapName " ") (colFact -1)
(hasSQLObjects "firstname") (htmlFactID -1) (hasHTMLObj "Input") (htmlName " ")
(weight 0) (value " ") (isPartOf "scholarships_form"))
==> Activation: MAIN::updMapObj :  f-18, f-22, f-21
FIRE 1 MAIN::updMapObj f-18, f-22, f-21
 <=> f-22 (MAIN::MapObjects (objName "First Name") (mapName "FIRSTNAME") (colFact
18) (hasSQLObjects "FIRSTNAME") (htmlFactID 21) (hasHTMLObj "Input") (htmlName
"firstname") (weight 0) (value " ") (isPartOf "scholarships_form"))
==> Activation: MAIN::updateDescription :  f-18, f-22
FIRE 2 MAIN::updateDescription f-18, f-22
<== Activation: MAIN::checkDatatype :  f-18,
 <=> f-18 (MAIN::SQLColumns (tableName "GCSCHOLARSHIPS_FORM") (colName
"FIRSTNAME") (colType "zzz") (size 128) (weight 0) (description "First Name"))
==> Activation: MAIN::checkDatatype :  f-18,
==> Activation: MAIN::checkColInTable :  f-18,
<== Activation: MAIN::checkColInTable :  f-18,
FIRE 3 MAIN::checkDatatype f-18,
Type zzz is not a valid type. Changing to default "varchar2".
 <=> f-18 (MAIN::SQLColumns (tableName "GCSCHOLARSHIPS_FORM") (colName
"FIRSTNAME") (colType "varchar2") (size 128) (weight 0) (description "First
Name"))
==> Activation: MAIN::checkColInTable :  f-18,
<== Activation: MAIN::checkColInTable :  f-18,
FIRE 4 MAIN::updateApps f-20
 ==> f-23 (MAIN::__query-trigger-getMapObjs "scholarships_form")
 <== f-23 (MAIN::__query-trigger-getMapObjs "scholarships_form")
 <=> f-20 (MAIN::Apps (hasDepartment "Gulf Coast Admissions") (appName
"scholarships form") (hasSQLTables "GCSCHOLARSHIPS FORM") (hasPart 22))
```

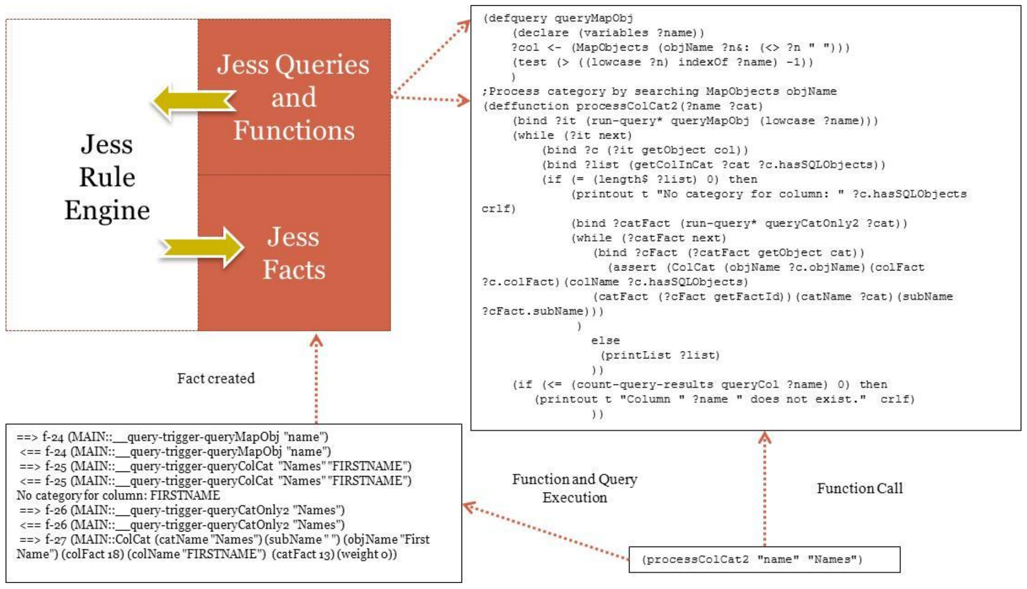
Figure 22b. Output of the Execution of Assert Statements for Department, Apps, Input, and MapObjects.

Figure 22b, shows the second half of the output of the same program input but showing the results of when assert statements for Department, Apps, Input, and MapObjects are executed and the activation and firing of the corresponding rules. As seen in the output of Figures 22a and 22b, the creation of a fact causes one or more rules to be activated and fired. Rule-based programs do not flow the same way as sequential and object-oriented programs. As stated earlier, rules in a rule-based program are executed upon the existing of facts matching the LHS of the rule.

The creation of an Apps fact cause the “updateApps” rule to be activated. However, this rule is fired after the corresponding MapObjects fact is created. The RHS of this rule calls the function “getObjects” and through a query it retrieves and returns the MapObjects fact IDs that have a value in the “isPartOf” slot that matches the value in the “appName” slot of the Apps fact. The “updateApps” rule updates the “hasPart” slot in the Apps fact with the MapObjects fact ID number returned by the function. As stated earlier, each fact in the Jess working memory is associated to an ID. The fact ID is useful when there is a need to refer to the fact. Using the fact ID as a reference to the fact is a practical way to identify the fact.

Another rule activated is the “updMapObj”. This rule is activated and fired after a fact of MapObjects type is created and SQLColumns and FormTagsType facts exist and match the criteria on the LHS of the rule. The RHS of this rule modify the MapObjects fact by updating the following slots: mapName, htmlFactID, colFact, and htmlName. After the MapObjects fact is updated, the “updateDescription” rule is activated and fired. The existence of a SQLColumns fact and the existence of a MapObjects fact with the same value in the “colName” and “hasSQLObjects” slots, respectively, match the LHS of the rule and initiate the RHS of this rule to be carried out. The RHS of this rule modify the value in the “description” slot of the SQLColumns fact with the same value in the “objName” slot of the MapObjects fact. The “colFact” slot in the MapObjects fact is updated with the fact ID of the SQLColumns fact is also updated in this rule. Because the SQLColumns fact was modified, the “checkDatatype” and “checkColInTable” are activated and fired as necessary. If the “colType” slot in the SQLColumns fact does not have a valid datatype, the “checkDatatype” is fired and the slot is updated with the default type “varchar2.” After this update, any other rule dependent on SQLColumns are activated and fired if necessary.

Continuing with the flow of the program, the next rule to be activated is the “checkColInTable” rule. This rule is activated because of the changes that occurred in the “colType” slot of the SQLColumns fact. The rule is not fired, and it is deactivated due to no changes in the table to which this SQLColumn fact belongs to. Finally, the last rule to be fired is the “updateApps” rule. After the Apps fact is updated, the focus is turned to creating categories for the facts. A function was defined to query any MapObjects fact that has a value in the “objName” slot that matches a given name. The list of matching MapObjects facts is processed, and any matching fact is utilized in creating categories for a given category name. For instance, the call to the function “processColCat2” would pass two arguments: 1) the name of the matching fact, such as “name” and 2) the name of the proposed category, such as “Names”. The query searches the value in the “objName” slot of each MapObjects fact that has “name” as part of the value. If a MapObjects fact is found, the corresponding SQLColumn fact for that MapObjects fact is processed for the “Names” category leading to the creation of a ColCat fact for the “Names” category. Figure 23 summarizes the process of creating ColCat fact described here.



```
(defquery queryMapObj
    (declare (variables ?name))
    ?col <- (MapObjects (objName ?n&: (<> ?n " ")))
    (test (> ((lowcase ?n) indexOf ?name) -1))
    )
;Process category by searching MapObjects objName
(deffunction processColCat2(?name ?cat)
    (bind ?it (run-query* queryMapObj (lowcase ?name)))
    (while (?it next)
        (bind ?c (?it getObject col))
        (bind ?list (getColInCat ?cat ?c.hasSQLObjects))
        (if (= (length$ ?list) 0) then
            (printout t "No category for column: " ?c.hasSQLObjects
crlf)
            (bind ?catFact (run-query* queryCatOnly2 ?cat))
            (while (?catFact next)
               (bind ?cFact (?catFact getObject cat))
                 (assert (ColCat (objName ?c.objName)(colFact
?c.colFact)(colName ?c.hasSQLObjects)
               (catFact (?cFact getFactId))(catName ?cat)(subName
?cFact.subName)))
            )
              else
               (printList ?list)
              ))
    (if (<= (count-query-results queryCol ?name) 0) then
       (printout t "Column " ?name " does not exist."  crlf)
              ))
```

```
==> f-24 (MAIN::__query-trigger-queryMapObj "name")
 <== f-24 (MAIN::__query-trigger-queryMapObj "name")
 ==> f-25 (MAIN::__query-trigger-queryColCat "Names" "FIRSTNAME")
 <== f-25 (MAIN::__query-trigger-queryColCat "Names" "FIRSTNAME")
No category for column: FIRSTNAME
 ==> f-26 (MAIN::__query-trigger-queryCatOnly2 "Names")
 <== f-26 (MAIN::__query-trigger-queryCatOnly2 "Names")
 ==> f-27 (MAIN::ColCat (catName "Names") (subName " ") (objName "First
Name") (colFact 18) (colName "FIRSTNAME") (catFact 13) (weight 0))
```

```
(processColCat2 "name" "Names")
```

Figure 23. ColCat Fact Creation Process.

In addition to the rules defined so far, other rules were defined to process the facts. The example provided only shows the creation of a fact that is both a HTML item and a SQL column item. What if the item is only an HTML item without being a SQL column? An example of this situation is when there is a submit button on the Web-based form. This button is an HTML item and does not exist as a SQL column item. What if the item is only a SQL column item without being an HTML item? An example of this scenario is when there is a column for storing the current date and time the Web-based form was submitted. This type of item does not need to exist as an HTML item. Rules for these types of items were created. As formerly stated, all items have a related MapObjects fact. The MapObjects fact contains slots that identify all items whether the items are only HTML items, SQL items, or both. Table 8 shows a summary of the rules defined in the program in addition to the rules defined in Table 7.

Table 8

Rules Defined for Updating or Asserting Facts

Rule Name	Templates	Modify/Assert	Slots	Description
checkColExist4Input	Input, SQLColumns, Apps	Input	hasSQLObjects	Condition: this slot is blank; rule modifies this slot with the matching column name that is a column in a table that is part of an application, Apps. The tableName slot value in SQLColumns matches a value in the hasSQLTables slot in the Apps fact
checkColExist4Select	Select, SQLColumns, Apps	Select	hasSQLObjects	Condition: this slot is blank; rule modifies this slot with the matching column name that is a column in a table that is part of an application, Apps. The tableName slot value in SQLColumns matches a value in the hasSQLTables slot in the Apps fact
updMapObj	SQLColumns, MapObjects, FormTagsType	MapObjects	mapName, htmlFactID, hasHTMLObj, colFact, hasSQLObjects, htmlName	Modifies slots with the matching column name; the value in colName is matched with the value in hasSQLObjects of both MapObjects and FormTagsType facts
updMapObj2	FormTagsType, MapObjects	MapObjects	mapName, htmlFactID, hasHTMLObj, htmlName	Modifies slots with the matching HTML object; this MapObjects fact does not have a column associated; it is just an HTML item that is not in a SQL Table

Table 8 (continued).

Rule Name	Templates	Modify/Assert	Slots	Description
updMapObjOptgroup	FormTagsType, MapObjects	MapObjects	objName, mapName, htmlFactID, htmlName	Modifies slots with the matching HTML object that is an "Optgroup" element; the objName slot is updated with the value in the label slot of the Optgroup fact
updMapObjSQLNoHTML	SQLColumns, MapObjects, Apps	MapObjects	objName, mapName, hasSQLObjects, isPartOf	This MapObjects fact is not an element in a form; this is a SQLColumns fact; the slots are modified with the matching values found in the SQLColumns fact; the Apps appName is the value assigned to isPartOf slot
updateApps	Apps	Apps	hasPart	hasPart is updated with a list of MapObjects fact ID that has the appName of this Apps fact; a function is called to query working memory and find the matching MapObjects
updateDescription	SQLColumns, MapObjects	SQLColumns, MapObjects	description (SQLColumns), colFact (MapObjects)	Condition: The value in objName slot of MapObjects fact is not blank and the hasSQLObjects value matches the name of the colName slot in SQLColumns; description slot is updated with same value in objName and colFact is updated with the SQLColumns fact ID

Table 8 (continued).

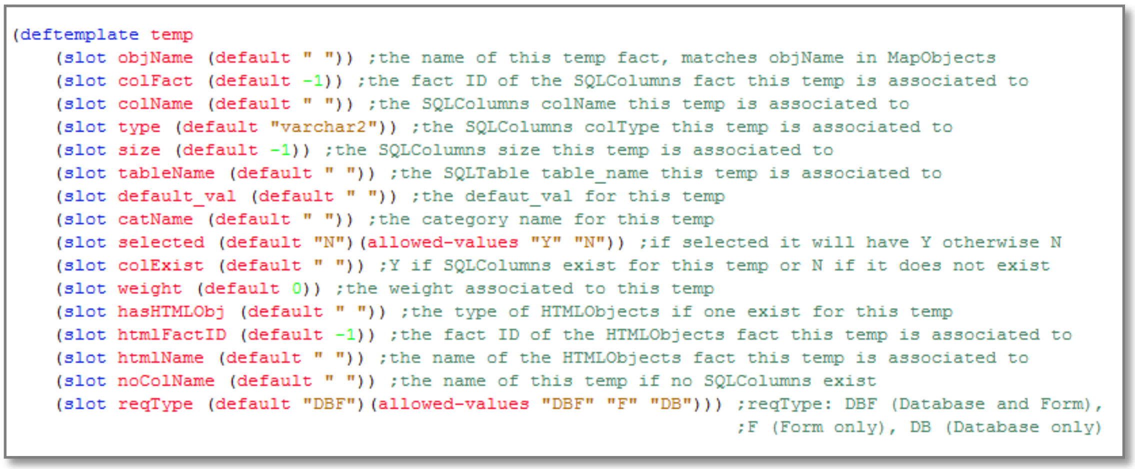
Rule Name	Templates	Modify/Assert	Slots	Description
updateDescription2	FormTagsType, MapObjects	FormsTagType	description	Condition: description slot is blank; the type of this FormsTagType fact matches the value in hasHTMLObj slot in MapObjects; the fact ID of this formsTagType fact matches the fact ID in htmlFactID slot of MapObjects fact; the description slot is updated with the value in objName of MapObjects
createMapObject	SQLColumns, FormTagsType	MapObjects		Assert a MapObjects fact that does not exist for the SQLColumns and FormTagsType facts.
updateDescription3	MapObjects	MapObjects	objName	Condition: objName slot is blank; mapName slot contains a value; this fact does not have a value in hasHTMLObj; the objName slot is updated with the value in mapName

The rules, as summarized in Table 8, are executed for each fact that matches the LHS of the rule. The description column details the condition of the rule and what is executed on the RHS of the rule. On the RHS of the rule, an existing fact may be modified or a new fact may be asserted.

Upon the execution of all assert statements for the selected Web-based forms, all facts were created and saved to an external file in the order they were created. The save-facts command in Jess allows for all facts to be saved to a specified file. This file was saved in the same directory as the program and ready for loading when processing new requirements.

Rules for New Requirements

The additional templates, rules, functions, and queries implemented permit new customer requirements to be processed and analyzed for ambiguities. These rules are specific to processing customer requirements by allowing the user to define the requirements while using historical requirements data. The benefits of these rules let new requirements to be matched against existing requirements in the knowledge-base. When a new requirement is entered, specific rules, functions, and queries are fired and a “temp” fact is created for the requirement. This new “temp” fact is created through the “newReq” rule and “processNewReq” function. The “temp” fact contains slots that associate this fact to all other facts in the knowledge-base. Figure 24 shows the definition for the “temp” template.



```
(deftemplate temp
    (slot objName (default " ")) ;the name of this temp fact, matches objName in MapObjects
    (slot colFact (default -1)) ;the fact ID of the SQLColumns fact this temp is associated to
    (slot colName (default " ")) ;the SQLColumns colName this temp is associated to
    (slot type (default "varchar2")) ;the SQLColumns colType this temp is associated to
    (slot size (default -1)) ;the SQLColumns size this temp is associated to
    (slot tableName (default " ")) ;the SQLTable table_name this temp is associated to
    (slot default_val (default " ")) ;the defaut_val for this temp
    (slot catName (default " ")) ;the category name for this temp
    (slot selected (default "N")(allowed-values "Y" "N")) ;if selected it will have Y otherwise N
    (slot colExist (default " ")) ;Y if SQLColumns exist for this temp or N if it does not exist
    (slot weight (default 0)) ;the weight associated to this temp
    (slot hasHTMLObj (default " ")) ;the type of HTMLObjects if one exist for this temp
    (slot htmlFactID (default -1)) ;the fact ID of the HTMLObjects fact this temp is associated to
    (slot htmlName (default " ")) ;the name of the HTMLObjects fact this temp is associated to
    (slot noColName (default " ")) ;the name of this temp if no SQLColumns exist
    (slot reqType (default "DBF")(allowed-values "DBF" "F" "DB"))) ;reqType: DBF (Database and Form),
                                                                   ;F (Form only), DB (Database only)
```

Figure 24. Template Definition for “temp” and its Slots.

The “processNewReq” function searches the current working memory for MapObjects facts matching the value in the “objName” slot with the newly input requirement. The “temp” fact for this requirement is created whether or not a fact is found. The procedure for handling new requirements as seen in Figure 25 shows the two different paths: 1) the newly input requirement matches an existing fact in the historical knowledge-base, and 2) the newly input requirement does not have a matching fact in the historical knowledge-base.

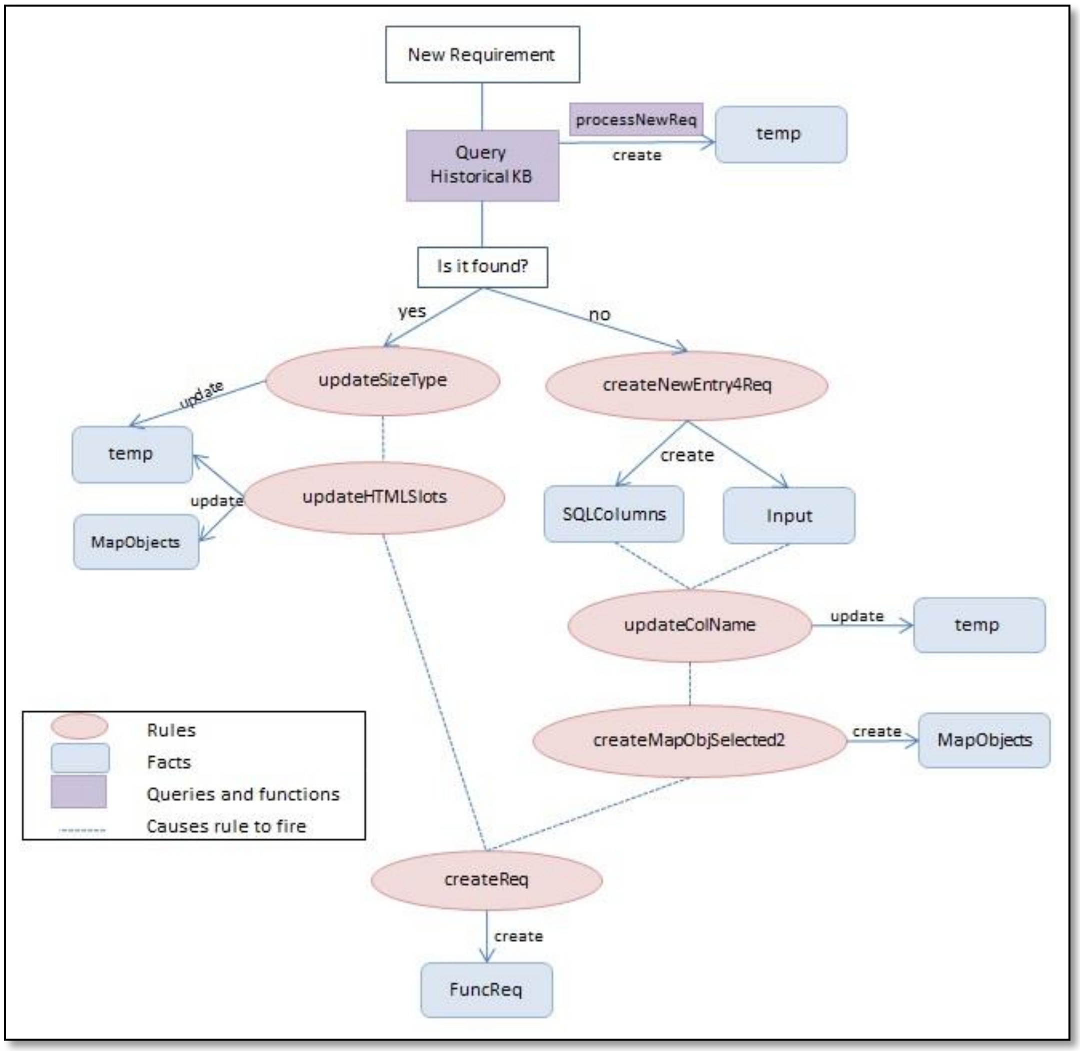


Figure 25. Process Flow for Handling New Requirement.

If the fact is found, the corresponding slots in the "temp" fact for this new requirement are updated via the "updateSizeType" rule. The "objName" slot is updated with the same value as in the "objName" slot of the MapObjects fact. If the matching MapObjects fact has a corresponding SQLColumns fact, the "colFact", "colName", "type", "size", and "tableName" slots are populated with the same values of the matching SQLColumns fact. The temp fact is also updated through the "updateHTMLSlots" rule. If the matching MapObjects fact has a corresponding HTMLObjects fact, the "hasHTMLObj", "htmlFactID", and "htmlName" slots are populated with the same values of the matching HTMLObjects fact. If a category exists for the MapObjects, the "catName" slot is populated with the corresponding category. The "selected" slot is populated with a "Y" if this "temp" fact is selected as a requirement to be included in the initial draft, or "N" if otherwise. If the "colExist" slot value is a "Y", then a SQLColumns exist for this "temp" fact, otherwise, a value of "N" is in this slot and the "noColName" contains the name of the requirement entered. The "reqType" slot is used to store the type this requirement is used in the corresponding application. If the requirement is to be shown on the form front end and backend, in the database, then the value of this slot will be "DBF", otherwise, it will have a value of "F" for form only, and "DB" for database only. Finally, the "weight" slot is used to include the weight of this requirement and how often it is applied to other applications. The weight slot also allows for sorting requirements and grouping requirements that are used often together. The idea for using weight is considered in future applications of the proposed process.

If no matching is found for the newly input requirement, the “temp” fact is created with only the “objName” and “noColName” slots being populated. These slots contain the name of the requirement entered. When this is the case, certain rules and functions are executed in order to create SQLColumns, HTMLObjects, and MapObjects facts for this new requirement. The first rule to be executed is the “createNewEntry4Req”. This rule is fired, and a SQLColumns fact and an Input fact are created. These facts are created by default to populate the slot with given and default values. The slots in these facts are populated with default values suggested by the system and later updated as per user request. The default slot values populated in the SQLColumns fact and in the Input fact are summarized in Figure 26.

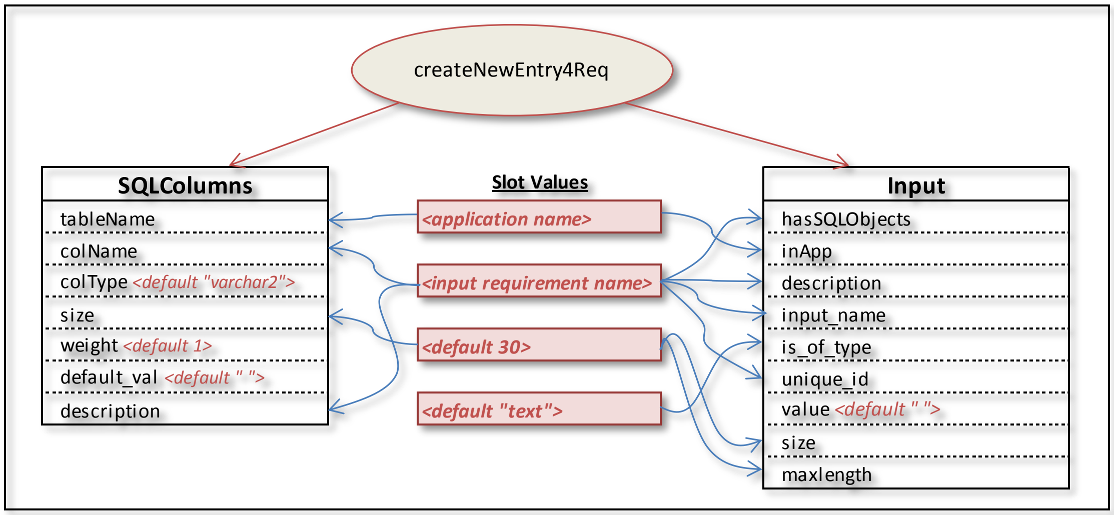


Figure 26. SQLColumns and Input Facts and the Values that go into each Slot upon the Execution of the createNewEntry4Req Rule.

Each slot in the corresponding fact is populated with the slot values shown or with the default value of the slot. The given application name at the time of requirement input fills the “tableName” and “inApp” slots of the SQLColumns and Input facts respectively. The “colName” and “description” slots of the SQLColumns fact and the “hasSQLObjects”, “description”, “input_name”, and “unique_id” slots of the Input fact are populated with the name of the input requirement. The default size and maxlength is set to 30. This default value is suggested by the system and may be modified as per user request. Text is the default type of the Input fact created. All other slots are populated with the default value of the slot.

After the SQLColumn and Input facts are created, the temp fact is updated with the correct slot values for the SQLColumns fact and the Input fact. This update is carried out through the “updateColName” rule. The MapObjects fact is then instantiated through the “createMapObjSelected2” rule. Figure 27 shows the values for each slot in the MapObjects fact for the new requirement.

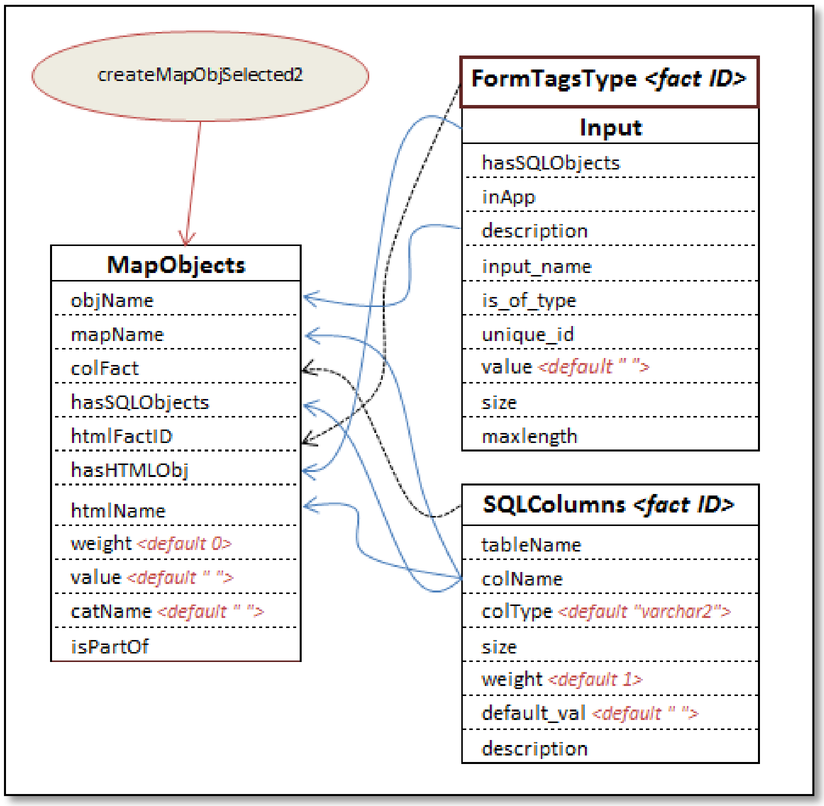
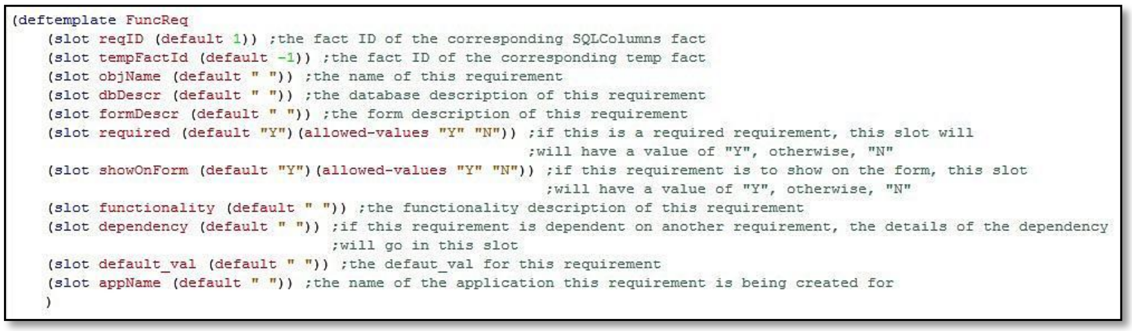


Figure 27. MapObjects Fact and Slot Values after createMapObjSelected2 Rule Execution.

The “createMapObjSelected2” rule asserts a fact of MapObjects based on the SQLColumns and Input facts created. The slots in the newly created MapObjects fact are populated as follow. The “objName” slot takes the same value as the value in the “description” slot in the Input fact. The SQLColumns “colName” slot value populates the “mapName”, “hasSQLObjects”, and “htmlName” slots. The “colFact” slot is populated with the fact ID of the SQLColumns fact. The “htmlFactID” slot takes the fact ID of the parent class of the Input template, in this case the FormTagsType fact ID. The “hasHTMLObj” slot is populated with the name of this FormTagsType, which is “Input.” All other slots are populated with default values as defined in the MapObjects template.

Finally, after the creation of a MapObjects fact and the temp fact is updated, a fact for the new requirement is created through the “createReq” rule. This rule creates a fact of FuncReq. The FuncReq template, as depicted in Figure 28, defines the structure of a functional requirement for the domain being studied in this dissertation.



```
(deftemplate FuncReq
    (slot reqID (default 1)) ;the fact ID of the corresponding SQLColumns fact
    (slot tempFactId (default -1)) ;the fact ID of the corresponding temp fact
    (slot objName (default " ")) ;the name of this requirement
    (slot dbDescr (default " ")) ;the database description of this requirement
    (slot formDescr (default " ")) ;the form description of this requirement
    (slot required (default "Y")(allowed-values "Y" "N")) ;if this is a required requirement, this slot will
                                                          ;will have a value of "Y", otherwise, "N"
    (slot showOnForm (default "Y")(allowed-values "Y" "N")) ;if this requirement is to show on the form, this slot
                                                            ;will have a value of "Y", otherwise, "N"
    (slot functionality (default " ")) ;the functionality description of this requirement
    (slot dependency (default " ")) ;if this requirement is dependent on another requirement, the details of the dependency
                                    ;will go in this slot
    (slot default_val (default " ")) ;the defaut_val for this requirement
    (slot appName (default " ")) ;the name of the application this requirement is being created for
    )
```

Figure 28. FuncReq Template Definition and its Slots.

The “createReq” rule creates a FuncReq fact based on facts matches between temp, MapObjects, and FormTagsType facts. These facts must have values in some of the slots that are common in all 3 facts. The “objName” slot in the temp fact must match the “objName” slot of the MapObjects fact. The name of the column stored in the “colName” slot of the temp fact need to be equal to the values in the “mapName” and “hasSQLObjects” slots of the MapObjects fact. The “colFact” slot in both temp and

MapObjects facts are also coordinated. The fact ID of the FormTagsType fact corresponding to this match is also coordinated with the “htmlFactID” slot in the temp and MapObjects facts. The name of one of the subclasses of the FormTagsType corresponds to the value in the “hasHTMLObj” slot in both MapObjects and temp facts. The value in the “htmlName” slot in both temp and MapObjects facts must also match. As stated earlier in this dissertation, the matching of select slots and facts on the LHS of the rule must take place before the RHS of the rule is executed. Once a matching occurs, the RHS of the rule is carried out and additional processing is followed. Figure 29 summarizes the process performed by the “createReq” rule.

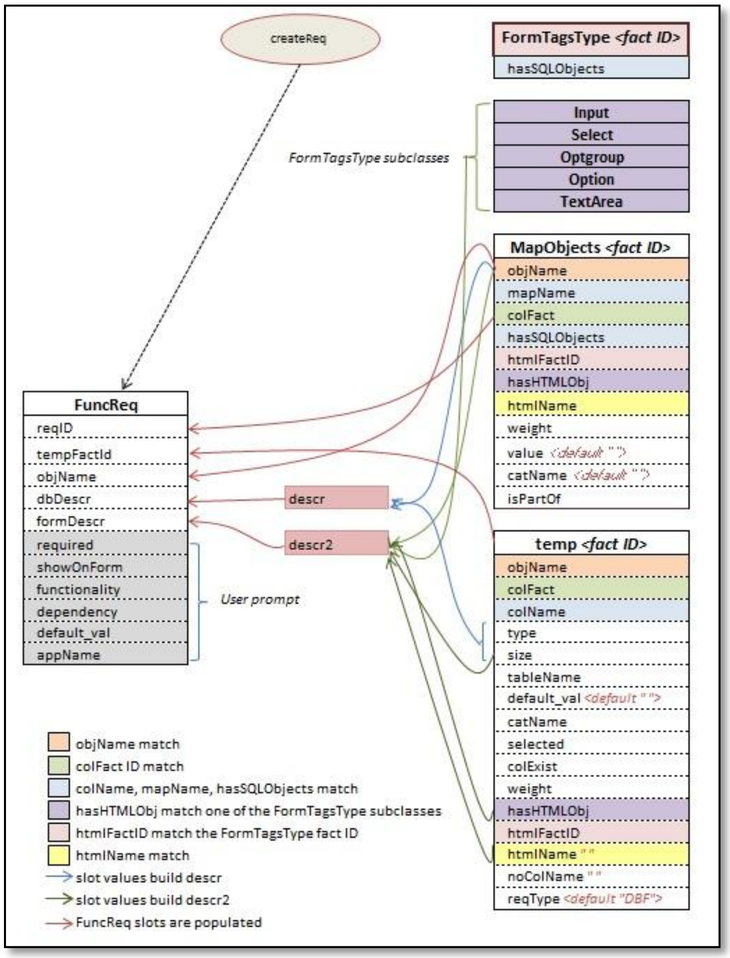


Figure 29. Rule for the “createReq” Process.

The “descr” and “descr2” elements shown in Figure 29 correspond to variables that are utilized as placeholders for the description of the database and form respectively slots of the FuncReq fact. The “descr” variable is built using the values of the slots pointed by the blue arrows, which include the “objName” slot of the MapObjects fact and the “type” and “size” slots of the temp fact. The description of the new requirement as it should be on the form is stored in the “descr2” variable. The “descr2” variable is constructed using the value of the slots pointed by the green arrows, which comprise of the name of one of the FormTagsType subclasses, the value in the “objName” slot in the MapObjects fact, and the value of the slots “size”, “hasHTMLObj”, and “htmlName” of the temp fact. The values of the slots in the FuncReq fact are populated through the red arrows. The “required”, “showOnForm”, “functionality”, “dependency”, and “appName” slot values are filled through user input.

Following the creation of the FuncReq fact, all created facts are added to the historical knowledge-base and saved. The preliminary requirements document is also generated and the Preliminary Customer Requirement form is updated. This document contains the system’s suggested requirements. The default requirement sentence for any requirement processed is shown in Figure 30.

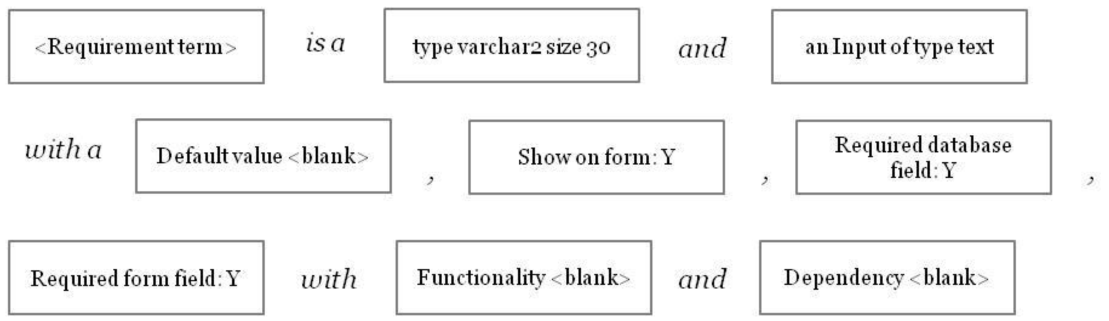


Figure 30. The Default Requirement Sentence for any Requirement Processed.

Any requirement that needs to be revised or modified can be processed in the system through the refinement procedure. In the refinement procedure, the corresponding area department and application must be known. Once the department and application names are entered in the system and the option to refine is chosen, the corresponding historical facts are loaded into working memory. The requirement to be refined is then input. The “refineRequirement” rule is activated and fired if the LHS of this rule finds a matching of the requirement to be refined on a MapObjects fact with a matching temp fact and a matching FuncReq fact as shown in Figure 31.

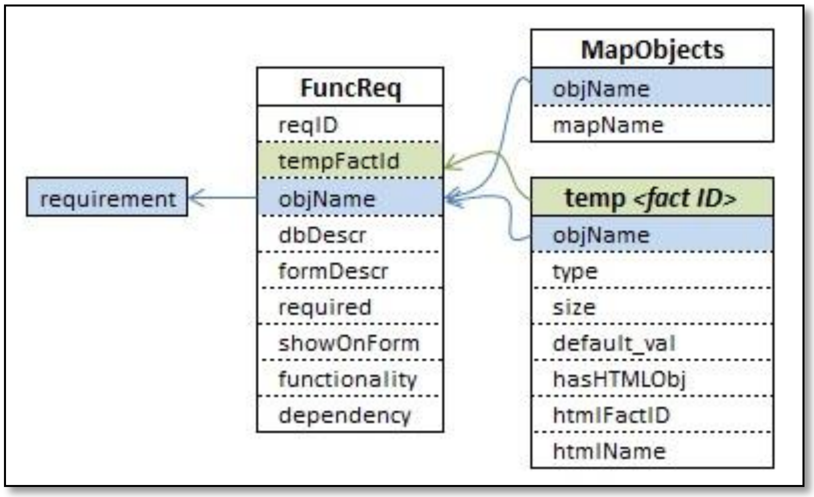


Figure 31. The LHS of the refineRequirement Rule.

As shown in Figure 31, there must be a match between the requirement to be refined and the “objName” slot value of the FuncReq, MapObjects, and temp facts before the rule can be activated and fired.

The RHS of the “refineRequirement” rule displays the details about the requirement, prompts the user to choose what to update, and through additional functions the requirement is updated. There are two additional functions that are important in the refinement process: 1) the “refCol” function allows the SQLColumns fact corresponding to this requirement to be updated, and 2) the “refHtml” function allows the corresponding FormTagsType fact to be updated. The updates on these facts cause the existing FuncReq fact to be retracted and a new fact generated. If there is no need to update the SQLColumns fact or the FormTagsType fact, the system prompts the user to update one or all of the following FuncReq slots values: “required”, “showOnForm”, “functionality”, and “dependency.” These slots are updated through functions. In each function, the user is prompted to enter the value for the corresponding slot. The value is returned to the “refineRequirement” rule, and the equivalent slots are updated in the FuncReq fact. Figure 32 shows the flow of this process carried out on the RHS of the "refineRequirement" rule.

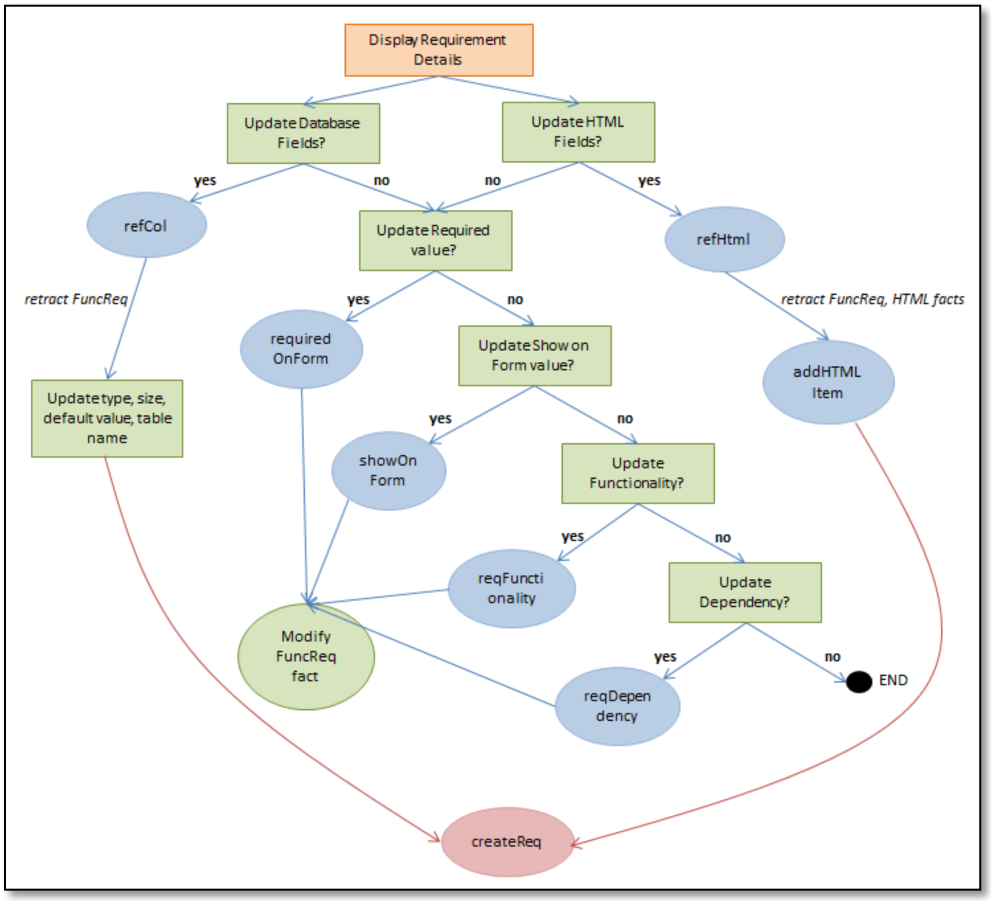


Figure 32. Flow of the RHS of the "refineRequirement" Rule.

The FuncReq fact needs to be retracted if the database fields and/or the HTML fields need to be updated because the "dbDescr" and "formDescr" slot values are constructed as strings in the "createReq" rule as shown in Figure 29.

CHAPTER V

CASE STUDY AND RESULTS

The results of the case study will in fact demonstrate whether or not the goal of this dissertation can be met in which requirements can be improved through reducing ambiguities with the use of a historical rule-based knowledge system while also improving the communication between customers, novice developers, and expert developers.

Selected Case Study

The selection of a case study included selecting subjects, environment, and a comparative requirement elicitation test for the study. The subjects selected for this study encompassed software developers of a software development unit. From here forth this software unit will be referred to as the IT unit. The software developers selected included a novice developer and an expert developer. A fictional customer was also selected for the test. The customer's main function was to provide a set of requirements for a new Web based application to be developed. The set of new requirements was used in a comparative test that was carried out in this study. The test carried out compares the results of reducing ambiguities through the process proposed in this study in two ways:

1. No historical data is available.
2. Historical data is implemented

In addition to carrying out this experiment with the set of new requirements in these two ways and comparing the results, an additional experiment with another set of requirements was also carried out. In the latter experiment, the requirements were processed using the historical data in knowledge-base, including history about the requirements of the first experiment.

The subjects of this study executed the two scenarios with the aid of the proposed scripted process and required forms. In both experiments, a customer provided the set of requirements and submitted them to the development team. The two sets of requirements explored in this experiment contained 27 and 15 requirement items, respectively. Appendix B shows the two sets of requirements utilized in this experiment. Each set of requirements are input separately into the Preliminary Customer Requirement form. Each set of requirements may be inconsistent and missing essential complementary requirements. As previously stated, a novice developer is in charge of processing these new requirements using the proposed process and identifying ambiguities in the requirements in order to improve the set of requirements. Due to the fact that a novice developer has vague knowledge about requirements elicitation, an experienced developer interacts with the novice developer when needed. The coordinated results between the user and the process are recorded in the Preliminary Customer Requirement form for each test. Ambiguities are evaluated and requirements needing refinement are processed again.

Each test scenario was processed through the proposed three-stage process, planning, processing, and evaluation, as discussed in Chapter III of this dissertation. The result of the tests performed in this experiment gives room for a discussion of the beneficial use of historical knowledge about the domain of Web-based applications when utilized for keyword matching.

Experiment and Results

Test Scenario 1: No Historical Knowledge

Planning phase. As shown in Figure 33, the first step of the scripted process to be completed is the planning phase. In this phase, the following are the the novice developer follows:

1. Retrieved the requirements from the customer
2. Input the requirements into the Preliminary Customer Requirement form in addition to any information about the Web-based application to be implemented. Additional information includes the name of the application, the purpose of the application, the department name for which the application is being built, the use of a database, and any known information about each requirement.

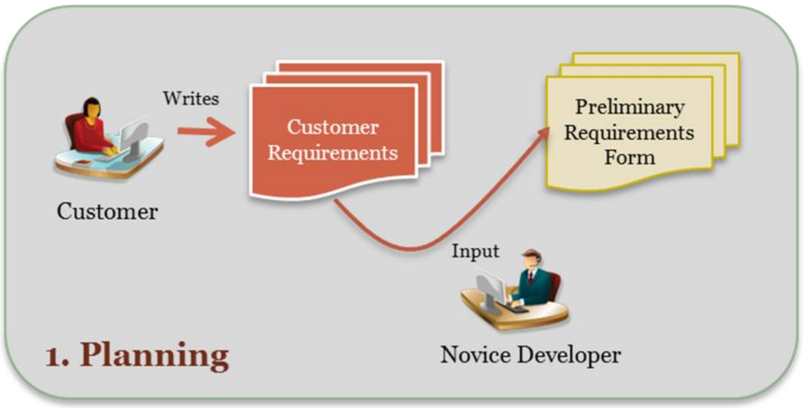


Figure 33. Planning Phase of the Step-by-Step Scripted Process.

Processing phase. In this next phase, as shown in Figure 34, the set of 27 requirements was processed without the knowledge of any previously defined requirement. Without any historical knowledge about requirements, the process at this point was only able to suggest a default requirement sentence for each requirement. The suggested sentence for each requirement was recorded into the same Preliminary Customer Requirement form.

3. List of requirements was input into the tool for processing
4. Tool suggested a list of requirement sentences for each requirement with default values for each term in the sentence (expert developer assisted novice developer as needed.)
5. Each requirement sentence was processed and ambiguities were detected.

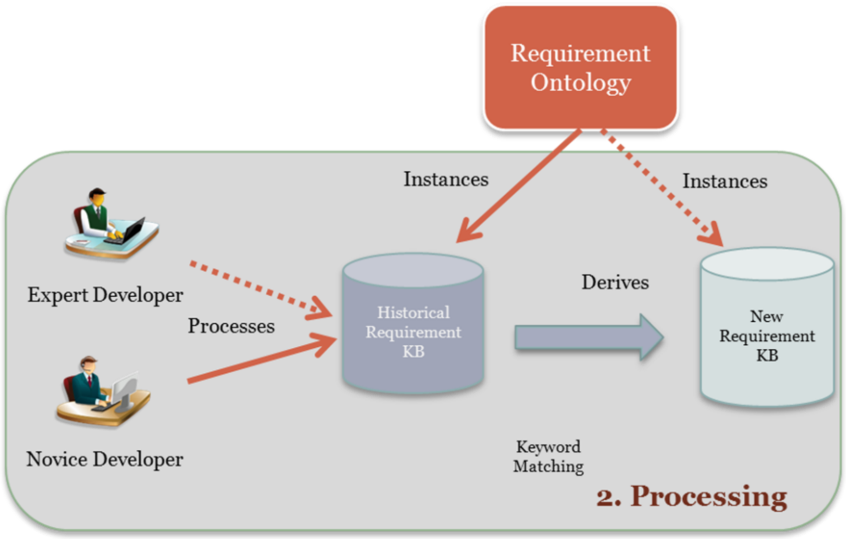


Figure 34. Processing Phase of the Scripted Process.

With the assistance of the expert developer, the novice developer was able to pinpoint the requirements that needed to be refined and the ones that needed complementary requirements. The findings of this first test were recorded as seen in Table 9 and the corresponding chart showing the percentage of ambiguous terms are shown in Figure 35.

Table 9

Results of First Test of 27 Requirements and No Historical Knowledge

Term	# of Ambiguities
Size	21
Datatype	6
HTML type	7
Functionality	7
Default value	7
Dependency	5

Table 9 (continued).

Improvement	# of Requirements
Unambiguous requirements	5
New requirements	0

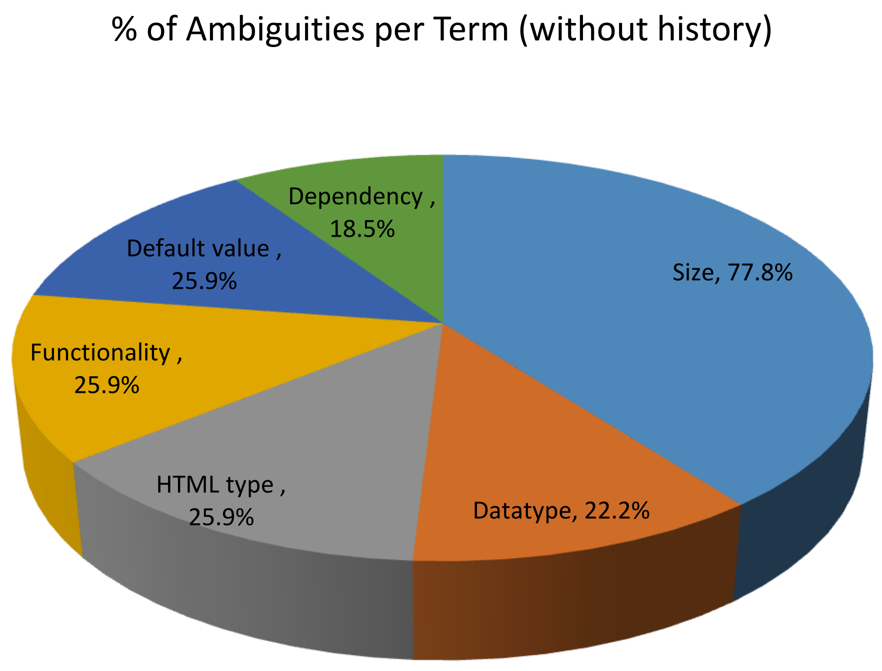


Figure 35. Percentage of Ambiguous Terms in All Produced Sentences.

As seen in Table 9, out of a total of 27 requirements, only five of those requirements were found to be unambiguous. This suggestion resulted from the interaction between the novice and expert developers after the requirements were processed. Due to the inexperience of the novice developer, there was a need for expert guidance in this first pass of the process. The unambiguity of a requirement does not mean the requirement is complete or that additional requirement is not needed. As previously stated, the unambiguity of a requirement is determined by the number of terms that need refinement in the requirement sentence. Table 9 also shows, according to the novice developer, the terms in the proposed requirement sentence that need to be refined. Out of the 27 requirements, 21 requirements need to have the term size revised, which corresponds to 77.8% of requirements, as seen in Figure 35. This is due to the fact that, as previously stated, the default size proposed is 30. Not all terms in the requirement sentence are listed in this table in order to avoid confusion and save space.

Since there was no history about previously defined requirements in the knowledge-base, this first test resulted in no additional requirements. Figure 36 shows the FuncReq facts created for the first 5 requirements processed.

```
(MAIN::FuncReq (reqID 26) (tempFactId 22) (objName "Social security number") (dbDescr
"Social security number is of type varchar2 size 30.") (formDescr "Social security
number will be an Input of type text.") (required "Y") (showOnForm "Y") (functionality
" ") (dependency " ") (default_val " ") (appName "app_without_history"))

(MAIN::FuncReq (reqID 42) (tempFactId 34) (objName "Email address") (dbDescr "Email
address is of type varchar2 size 30.") (formDescr "Email address will be an Input of
type text.") (required "Y") (showOnForm "Y") (functionality " ") (dependency " ")
(default_val " ") (appName "app_without_history"))

(MAIN::FuncReq (reqID 54) (tempFactId 49) (objName "Name") (dbDescr "Name is of type
varchar2 size 30.") (formDescr "Name will be an Input of type text.") (required "Y")
(showOnForm "Y") (functionality " ") (dependency " ") (default_val " ") (appName
"app_without_history"))

(MAIN::FuncReq (reqID 69) (tempFactId 61) (objName "Maiden Name") (dbDescr "Maiden
Name is of type varchar2 size 30.") (formDescr "Maiden Name will be an Input of type
text.") (required "Y") (showOnForm "Y") (functionality " ") (dependency " ")
(default_val " ") (appName "app_without_history"))

(MAIN::FuncReq (reqID 81) (tempFactId 76) (objName "Gender") (dbDescr "Gender is of
type varchar2 size 30.") (formDescr "Gender will be an Input of type text.") (required
"Y") (showOnForm "Y") (functionality " ") (dependency " ") (default_val " ") (appName
"app_without_history"))
```

Figure 36. FuncReq Facts Created for the First Five Requirements of the 27-Requirement Set.

As seen in Figure 36, all 5 requirements were created with the default values for the database description and HTML form description. The requirement *Name* was the only requirement is this subset that did not need to be revised. All other 4 requirements, shown here, needed to go through the refinement process.

Evaluation phase. Finally, during the evaluation phase, as shown in Figure 37, all requirements in need of refinement, that is all ambiguous requirements, were refined.

6. Each ambiguous requirement sentence was recorded in a separate Preliminary Customer Requirement form
7. Each requirement was input into the tool and refined as required
8. Suggested draft after all requirements were processed and refined is produced
9. Developer meets with customer to discuss findings
10 Repeat step 1 to 9 if necessary, otherwise, requirements can be included in specification document

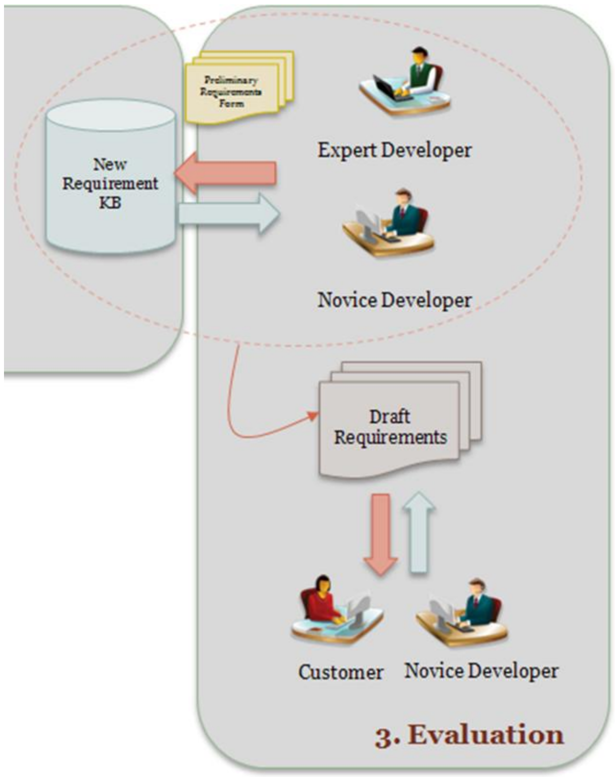


Figure 37. Evaluation Phase of the Proposed Scripted Process.

In this phase, the novice developer with the assistance of the expert developer went through each ambiguous requirement instance and made the necessary adjustments to each term of the requirement sentence as determined during the processing phase. During the refinement process, the requirements were redefined as shown in Figure 38.

```
(MAIN::FuncReq (reqID 36) (tempFactId 118) (objName "Social security number")
(dbDescr "Social security number is of type varchar2 size 9.") (formDescr "Social
security number will be an Input of type text.") (required "Y") (showOnForm "Y")
(functionality "Must be all numbers (e.g. 123456789)") (dependency "n/a")
(default_val " ") (appName "app_without_history"))

(MAIN::FuncReq (reqID 37) (tempFactId 146) (objName "Email address") (dbDescr "Email
address is of type varchar2 size 255.") (formDescr "Email address will be an Input of
type text.") (required "Y") (showOnForm "Y") (functionality "Must be in the format of
local-part@domain.com") (dependency "n/a") (default_val " ") (appName
"app_without_history"))

(MAIN::FuncReq (reqID 38) (tempFactId 120) (objName "Name") (dbDescr "Name is of type
varchar2 size 30.") (formDescr "Name will be an Input of type text.") (required "Y")
(showOnForm "Y") (functionality " ") (dependency " ") (default_val " ") (appName
"app_without_history"))

(MAIN::FuncReq (reqID 39) (tempFactId 121) (objName "Maiden Name") (dbDescr "Maiden
Name is of type varchar2 size 30.") (formDescr "Maiden Name will be an Input of type
text.") (required "Y") (showOnForm "Y") (functionality "Should have a value if the
gender field is an F") (dependency "gender") (default_val " ") (appName
"app_without_history"))

(MAIN::FuncReq (reqID 40) (tempFactId 122) (objName "Gender") (dbDescr "Gender is of
type char size 1.") (formDescr "Gender will be of type radio with the following
values:  Radio button: Gender value: F
Radio button: Gender value: M") (required "Y") (showOnForm "Y") (functionality "n/a")
(dependency "n/a") (default_val " ") (appName "app_without_history"))
```

Figure 38. Results of Refining the First Five Requirements.

As Figure 38 shows, the requirements in bold were the ones refined and the corresponding items in bold were also part of the refinement process of these first 5 requirements. The *Social security number* and *Email address* requirements had a change in the size and functionality terms, as shown. The *Maiden name* requirement had a change in its functionality and dependency sentence term. Finally, the *Gender* requirement had a change in its size and formDescr terms.

Once all requirements were refined, a draft of requirements was proposed and a meeting with the customer was set. The result of this meeting was not conclusive, and a final requirement draft was not produced. There was a need to include additional requirements in the original list of requirement, and additional iterations of the entire process were necessary. From this first test, it was determined that the addition of historical knowledge maybe essential to produce the desired list of requirements, which would include the additional requirements proposed by the customer.

Test Scenario 2: Historical Knowledge Exists

For the second test, historical knowledge was present and the same 27 requirements were processed. The planning, processing, and evaluation phases were basically the same, but with different results. The results of this test are shown in Table 10 and the corresponding pie chart in Figures 39 and 40.

Table 10

Results of Second Test of 27 Requirements with Historical Knowledge.

Term	# of Ambiguities
Size	15
Datatype	4
HTML type	7
Functionality	2
Default value	5
Dependency	3
Improvement	**# of Requirements**
Unambiguous requirements	11
New requirements	22

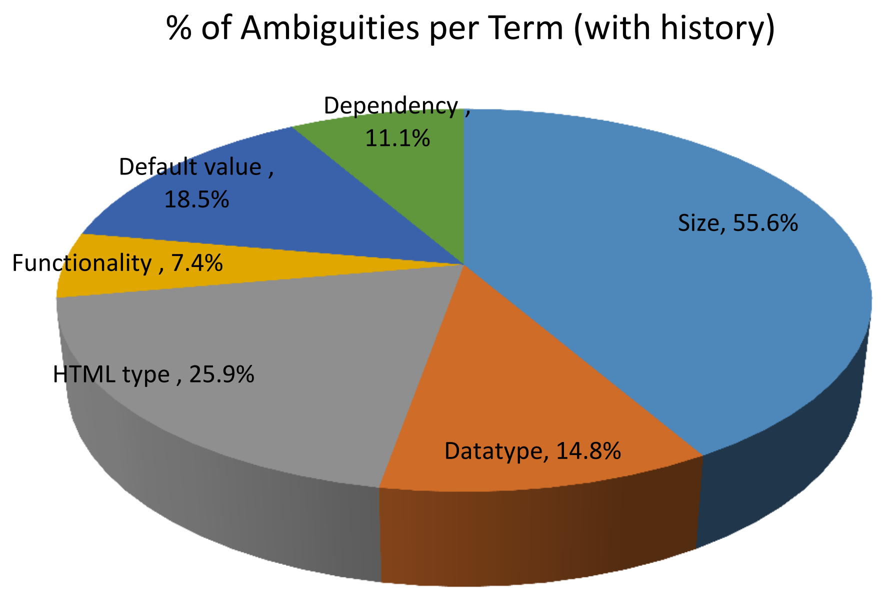


Figure 39. Percentage of ambiguous terms in all produced sentences.

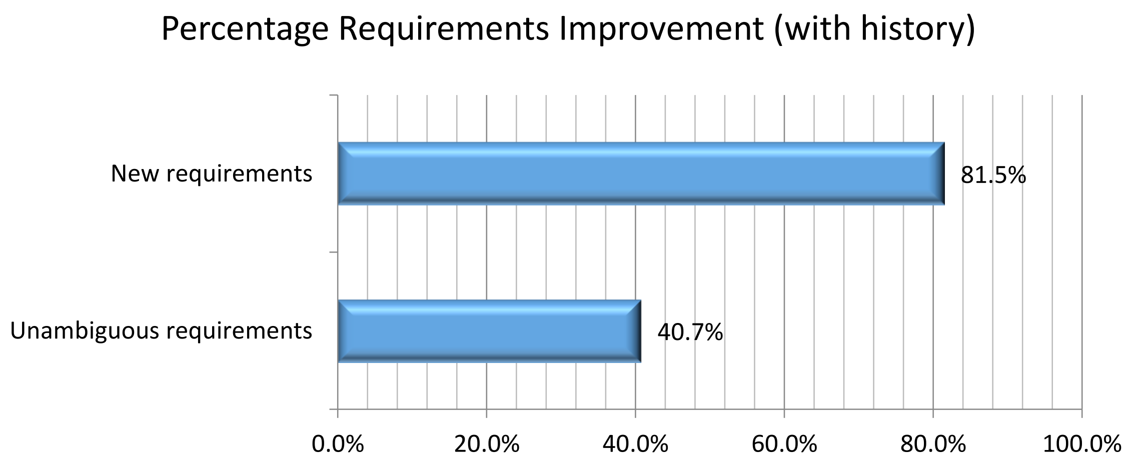


Figure 40. Percentage of Requirements Improvement with History.

The results shown in Table 10 after 27 requirements were processed using historical knowledge of previously defined requirements shows an improvement on the number of terms as compared to the results of the first test. When making use of history, it is important to see that new requirements were generated. These new requirements were produced in consequence of the matching that occurred between new and existing requirements. As shown in the chart presented in Figure 40, after the evaluation process, new requirements counted for 81.5% of the original list of requirements.

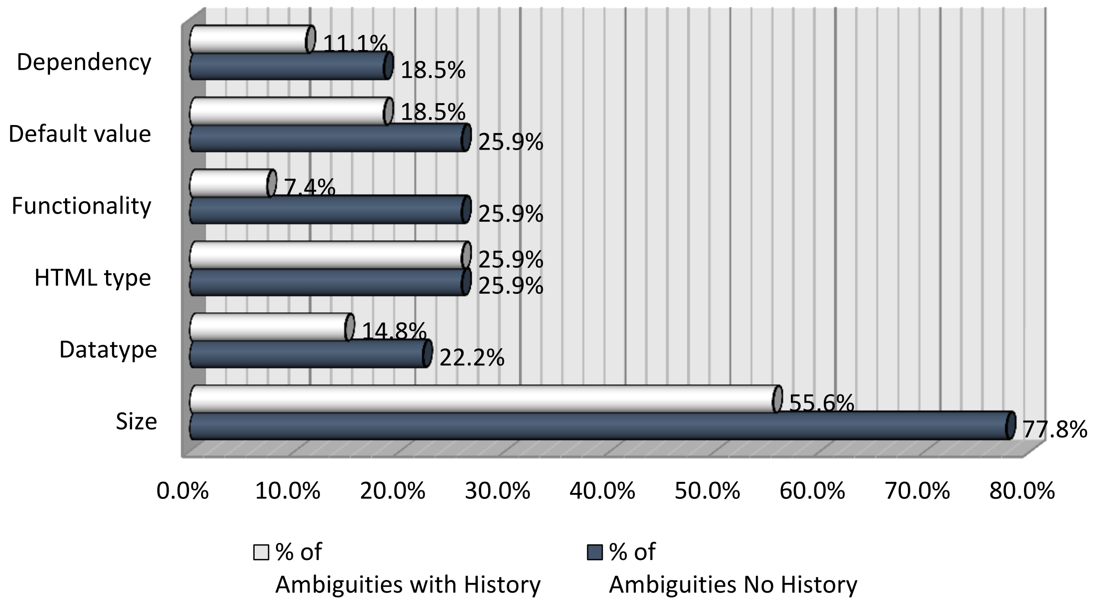
Because the process matches keywords in requirements, not all newly produced requirements were a perfect match for the requirements in the set. The developer had to manually map each new requirement to each corresponding requirement in the set. Basically, a requirement that had a matching requirement was not always the desired match. For example, the requirement “email address” in the set of processed requirements was found to be a match for the following existing requirements: contact_email, email, emailaddress, street address, other address, city, state, and zip. This match occurred because of the “email” and “address” words. Because the word “address” is associated to the category “Address”, all requirements in this same category were also matched against “email address.” Table 11 shows the requirements in the set that were produced and the number of requirements that were mapped.

Table 11

Requirements in the Set Mapped to the Newly Produced Requirements

Requirement in KB	Requirement in Set	Total # of New Requirements
	name	3
first name	maiden name	0
middle name	mother name	3
last name	father name	3
	child name	3
zip		
street address	email address	0
other address	permanent address	5
city	mailing address	5
state		
	Total:	**22**

As seen in Table 11, not all requirements in the set are mapped to a requirement matched in the knowledge-base. The "name", "mother name", "father name", and "child name" requirements each were mapped to "first name", "middle name", and "last name" requirements. The 4 requirements in the set were replaced by 12 new requirements. The "maiden name" requirement was not replaced by any requirement and remained in the requirement draft. The "email address" requirement was replaced by one of the suggested email requirements. The "permanent address" and "mailing address" were each mapped to "zip", "street address", "other address", "city", and "state" existing requirements and adding 10 new requirements to the requirements draft. After the manual mappings of the 8 matching requirements, a total of 22 new requirements were added to the original list of requirements, which correspond to the 81.5% increase in new requirements as compared to 0 new requirements in the first test without any historical knowledge. The comparison charts shown in Figure 41 shows where the addition of new requirements makes a visual difference to the number of requirements.



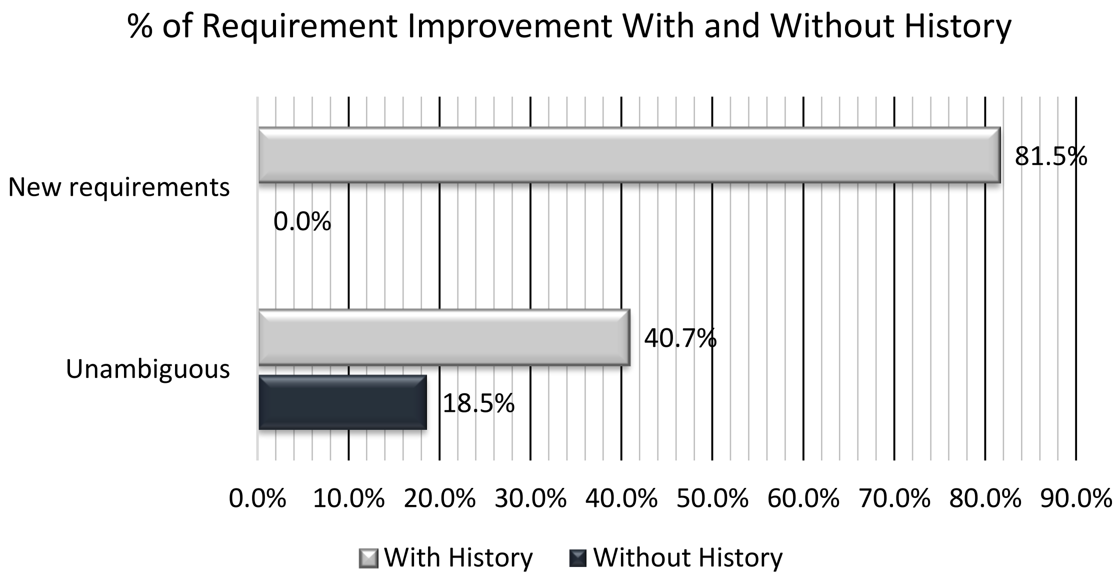


Figure 41. Comparison Charts Showing the Percentage Difference Between Test 1 and Test 2 Results.

From this second test, it was clear to see that in order to reduce the number of ambiguities in the requirements terms as shown in Table 10, there was a need to increase the number of historical facts in the knowledge-base. With additional history in the knowledge-base, it is implied that there would be a larger number of matches between requirements and fewer ambiguities among the terms in the proposed requirement sentence. Due to the limited size of the data being used in this test, a third test scenario was implemented.

Test Scenario 3: New Set of Requirements, Historical Knowledge Exists

In this third test scenario, a set of new requirements is employed. The set of requirements comprised of 15 customer requirements. The requirements were again listed as single statements as to what they would represent on a Web-based form. The process involved in this case study was very similar to the process carried out in the first set of requirements when the historical knowledge-base was available. The results of this scenario, as expected, is different from the results of the two scenarios, for the first set of requirements due to the number of requirements and the diverse types of requirements. The result of this test is shown in Table 12 and the corresponding chart in Figure 42 and 43.

Table 12

Results of Second Test of 15 Requirements with Historical Knowledge Present.

Term	# of Ambiguities
Size	5
Datatype	1
HTML type	2
Functionality	5
Default value	3
Dependency	1

Table 12 (continued).

Improvement	# of Requirements
Unambiguous requirements	10
New requirements	7

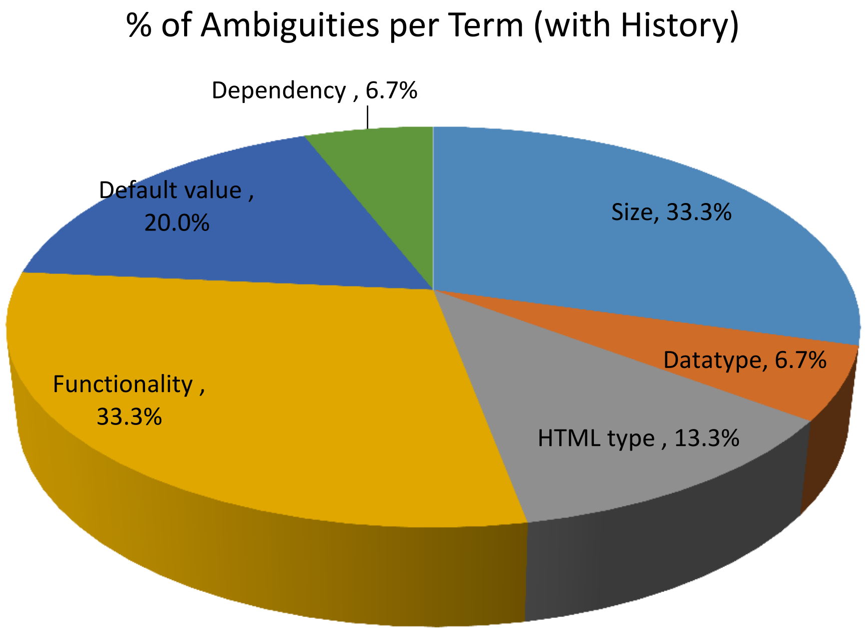


Figure 42. Percentage of Ambiguities per Term for the Second Set of Requirements when History is Present.

As shown in Table 12, the number of unambiguous requirements increased as compared to the first set of requirements. In this set of 15 requirements, 5 new requirements were created due to a match in the knowledge-base. As seen in Figure 42, this counted for 46.7% of the number of requirements that did not need refinement. The number of ambiguous terms in a sentence continued to show for the sentence terms size, datatype, HTML type, functionality, default value, and dependency due to the lack of matching between certain requirements. The automatic formation of a requirement sentence for the requirements that did not have a perfect match caused these terms to be formed with unexpected values.

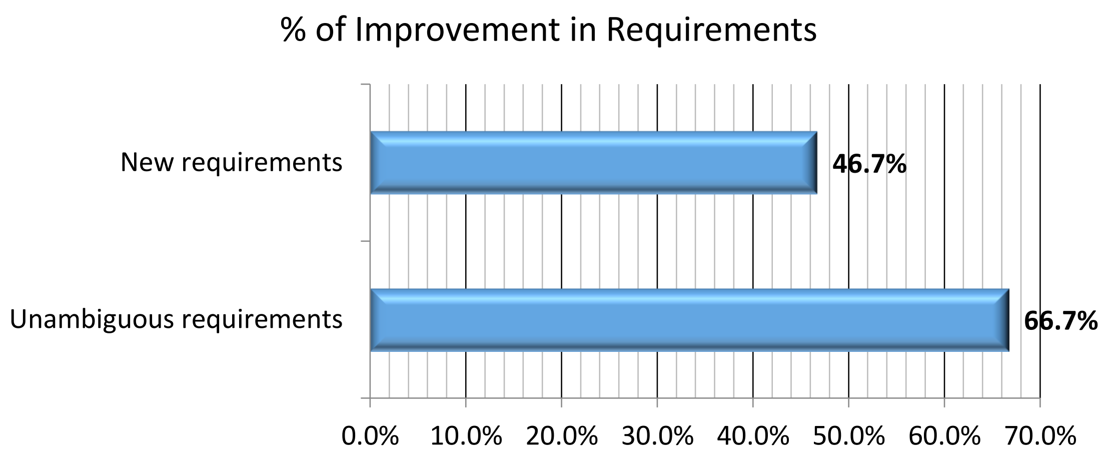


Figure 43. Improvement in Requirements when History is Available for the Set of 15 Requirements.

Another relevant aspect of this test includes the creation of new requirements. In this test, 8 new requirements were also produced. However, not all of these newly created requirements were taken into consideration. For instance, the requirement “preferred first name” caused the “first name”, “middle name” and “last name” requirements to be created. Because these 3 new requirements are not needed as per requirement set, there was no need to include these requirements. The “mailing address” and “hometown (city and state)” requirements were matched to “zip”, “street address”, “other address”, “city”, and “state” existing requirements in the knowledge-base. These requirements were considered and as shown in Table 13 they were mapped to each requirement as needed.

Table 13

Requirements in the Set Mapped to the Newly Produced Requirements

Requirement in KB	Requirement in Set	Total # of New Requirements
first name middle name last name	Preferred first name	0
zip street address other address city state	mailing address	5
	hometown (city and state)	2
	Total:	7

As seen in Table 13, the "preferred first name" requirement was created as a new requirement and did not use any of the matched requirements shown in the "Requirement in KB" column. The "mailing address" requirement was not created as a new requirement, but it was replaced by the 5 existing requirements: zip, street address, other address, city, and state. The "hometown (city and state)" requirement was replaced by the "city" and "state" requirement. These replacements gave a total of 7 new requirements that were added to the original set of requirements.

It is important to point out that the additional requirements added to the historical knowledge-base produced during the second case study, for the set of 27 requirements had an impact in the results of this scenario. Some of the requirements in this set of 15 requirements were the same or similar as the requirements in the first set. For instance, the "semester in which you intend to start" appears in both sets of requirements. In the first set, this requirement needed to be refined. Once it was refined and added to the historical knowledge-base, it became a match for the same requirement in the second set of requirements. As suggested before, as more requirements are added to the historical knowledge-base the better will be when it comes to matching future requirements and reducing ambiguities in the terms of the requirement sentence. As described, the first set of requirements produced better results when processed via the conceptual model when historical knowledge was present than the results when no historical knowledge was present. The second set of requirements also produced good results after it was processed. Having knowledge about the domain under discussion was the key in demonstrating how ambiguities in customer requirements can be reduced.

Even with the historical knowledge present, not all requirements had a perfect match. When no perfect match was found, the knowledge of the expert developer was essential in determining what parts of the requirement sentence needed attention. From experience in the area of Web-based form requirements, the expert developer was able to assist the novice developer in identifying the specific parts in the requirement sentence that demonstrated to be ambiguous. The size part of the requirement sentence was found to be the main part causing ambiguity in the requirement sentence in both sets of requirements. The size is often questionable as it depends on the type of field the requirement represents on the form.

Other requirements were not matched against similar requirements due to being worded differently. The unmatched requirements turn out to be ambiguous. For instance, the requirement “List the names of colleges you have attended” in the first set and the requirement “Current School OR School Last Attended” in the second set of requirements could have been matched if synonyms were employed as part of the matching process. However, both requirements are now in the knowledge-base, and any future new requirement matching either one will be generated.

In summary, the results after running two sets of requirements with and without historical knowledge did in fact demonstrate that ambiguities in customer requirements can be reduced and new requirements can be suggested. The results from the processed set of requirements in the case studies showed the need for knowing about the domain under discussion. Without knowledge about the requirements, no new requirements were suggested, and several requirements were found to be ambiguous. When knowledge about the domain was present, fewer requirements were ambiguous, and several new requirements were suggested.

Impact of Results

In addition to reducing ambiguities in customer requirements, the proposed semi-automated conceptual model was also able to suggest new requirements. It is also safe to say that in theory and by induction that the model was able to reduce the communication gap between the development team, both expert and novice developers, and between the development team and the customer. The potential combination of a rule-based tool and a scripted process imply the production of a less ambiguous set of customer requirements. Many studies have shown that a good set of functional requirements produces a good software product (Davis, Dieste, Hickey, Juristo, & Moreno, 2006; Herlea, Jonker, Treur, & Wijngaards, 1998; Jacobs, 2007; Jiang, Eberlein, & Far, 2004). Given the semi-automated process, it can be assumed that novice developers will be more knowledgeable of the domain under discussion and spend less time understanding the specified requirements. In addition, with the improved quality of customer requirements, one can assume better quality in the development effort and a reduction in the development time.

The results of this study have a direct impact on how customer requirements for Web-based forms are interpreted. It is well-known customers often do not know how to express their needs of what they want implemented. It is during this early stage of the requirement elicitation process that customer requirements are malformed and not understood by the people involved in the process, such those in the development team. With the use of a reasoning rule engine, historical knowledge-base of previously defined requirements and a step-by-step scripted process for requirements elicitation, customer requirements can become easier to understand. If requirements can become easier to understand, there will be fewer meetings scheduled between the developer and the customer. In the current setting where the testing took place, the developer involved needs time to review and to understand the set of customer requirements prior to meeting with the customer. The results of this dissertation can be summarized in terms of the developers and the set of requirements tested here are summarized in Table 14.

Table 14

The Effect of the Proposed Conceptual Model in Processing Requirements

Subject	Item	Before Conceptual Model	After Employing Conceptual Model
Expert Developer	Time spent reviewing and understanding the requirements	30 to 40 minutes	± 30 minutes
	Time spent meeting with the customer to understand requirements	1 to 1 ½ hours	± 30 minutes
	Number of meetings with customer	2 to 3	1 to 2

Table 14 (continued).

Subject	Item	Before Conceptual Model	After Employing Conceptual Model
Novice Developer	Time spent reviewing and understanding the requirements	1 ½ to 2 hours	± 30 minutes
	Time spent meeting with the customer to understand requirements	2+ hours	± 30 minutes
	Number of meetings with customer	2 to 3	1 to 2

As seen in Table 14, for someone with the skill set of an expert developer, it would take about 30 to 40 minutes just to review a set of 15 requirements prior to meeting with the customer as compared to the time using the conceptual model to just process the requirements. Then a meeting with the customer would be estimated to last from 1 to 1 ½ hours to examine the requirements. However, after using the conceptual model proposed here, this meeting may last about 30 minutes or even be eliminated. If only 5 requirements were identified to be ambiguous, the developer could contact the customer via email and avoid a meeting altogether.

The time reviewing the requirements and the time meeting with the customers for a novice developer varies slightly. For a novice developer, it would take 1 ½ to 2 hours reviewing the requirements with help from the expert developer due to the fact that a novice developer does not have the skills of an expert developer. After employing the conceptual model, this time is also reduced. The time a novice developer would spend meeting with a customer can be estimated to last 2 or more hours, but after employing the proposed conceptual tool, this meeting can last about 30 minutes or less. In this meeting, the novice developer would be accompanied by the expert developer in order to guide and answers questions a novice developer may not know the answer.

In addition to the first meeting with the customer taking place prior to development of requirements, a meeting during development and after development may be required. A meeting during development may become necessary. No matter if the process involves a novice or an expert developer, there will come a time that something will get missed. A missing element can impact the development timeline and a meeting with the customer is going to be required. A meeting with the customer after development may also be needed to run through everything to make sure all of the requirements were met. Or instead of a meeting, the customer should be able to go through the set of produced requirements and the application and to make sure the requirements were met.

In summary, the set of produced requirements can in fact reduce the communication gap between the developers and the customer and at the same time reduce ambiguities in customer requirements. The suggested new set of requirements and the improved requirements that are generated add knowledge to the domain. Both customer and novice developer become more acquainted with the overall process for eliciting requirements for Web-based form while reducing the communication gap. As compared to other studies, the results of the work explored in this dissertation have demonstrated to produce an impact in the requirements structure and definition. It has also caused an effect on how customer and developers communicate. As stated earlier, the study presented in Kaiya and Saeki (2006) the authors suggest a related technique for improving requirements, but the study does not take into consideration the effect of the technique on the people involved in the process and how they communicate.

CHAPTER VI

SUMMARY AND FUTURE WORK

This chapter summarizes the work presented in this dissertation and the contributions to the current research literature. It also presents the limitations of this study and potential future research related to the study and the results of this dissertation.

Summary

The objective of this study was focused on two obstacles during the elicitation of customer requirements: ambiguities in customer requirements for Web-based forms and communication gaps between customer and a novice developer. Research in the area of requirements elicitation process was accomplish and described in Chapter II. In Chapter III, a methodology was implemented to investigate and explore the implementation of a conceptual method and the scripted process to reduce ambiguities in customer requirements and bridge the communication gap between the people involved in the process. The focus of the proposed conceptual model was to improve misused and misunderstood parameters between domain experts and customers. After the creation of an ontology for Web-based forms, a knowledge-base of previously defined requirements was implemented and described in Chapter IV. The implemented knowledge-base was constructed using reasoning rules and a Java parser. The main function of the parser was to process existing Web-based forms to extract the requirement items that led to the development of the forms. Reasoning rules allowed for existing to be stored in the knowledge-base and allowed for these existing requirements and new requirements to be processed and matched via keywords. The results of the case studies utilized in this research were described in Chapter V. The results of the use of the proposed ontology and scripted process demonstrated the impact on how customer and developers communicate and how requirements are structured. In addition, it was important to keep track of the requirements as they were processed during the three-stage scripted process. The performance and evaluation of the conceptual method and of the scripted process are reasonably difficult to determine when method and process must act in conjunction.

As shown in Table 1 in Chapter II, this dissertation is linked with seven characteristics related to processing requirements. The characteristics of this dissertation in terms of the approach in this study are summarized in Table 15.

Table 15

Characteristics Approached in this Study

Dissertation Characteristics	Approach in this Study	Purpose
User experience	Novice developer, inexperienced customer, no analyst available; limited budget organization	Improve novice developer knowledge about the domain and how to process requirements
Ontology	Ontology based on requirements definition for Web-based form; conceptualization of HTML form elements and SQL table definition	Allow reusability of requirements, allow categorization of common requirements; define knowledge for Web-based forms: HTML and SQL create table
Rules for Reasoning about knowledge	Semi-automated reasoning using rule-based language allow keyword matching (Jess rules for reasoning about knowledge, keyword matching, and syntax processing)	Maintains integrity of requirements and allow reasoning about requirements
Scripted process and supported forms	Step-by-step procedure with supported forms to aid novice developer in reducing ambiguities in customer requirements	Reduce communication gap between people involved, adapts to domain, reduce meetings with customer
Historical Knowledge Base	Historical knowledge related to Web-based form requirements	Allow reusability of requirements, reduce ambiguities in requirements, improve requirement definition and allow unambiguous formation of requirement sentences

Through keyword matching using a rule-based programming language, it was possible to process customer requirements written in natural language. The creation of an ontology for categorizing the structure of Web-based form requirements made it possible to populate a knowledge-base of previously defined requirements. The previously defined requirements were categorized and organized based on their relationships. Newly defined requirements were matched against existing requirements using reasoning knowledge. The matching between new and existing requirements permitted the construction of a structured requirement sentence. The generated sentence for each requirement was evaluated, and ambiguities were identified. These steps were accomplished using the aid of a scripted process with instructions on how to process the requirements and how to identify ambiguities. The experience of the developers was important factors in this process. A novice developer was the main user of the process. When necessary, an expert developer assisted the novice developer during the process.

The ability to reuse requirements was one of the main characteristics of this dissertation. The suggested process improves new requirements by reusing previously defined requirements of formerly created Web forms. The effectiveness of the use of the conceptual model and the scripted process was established by the results of the comparative tests of two sets of new requirements. One test was executed using a set of 27 new requirements without any prior knowledge about the domain of Web-based forms. The same set was also tested using the conceptual model with available historical knowledge. The results of these tests were compared, and conclusions were drawn. The comparison results gave evident reasons to determine how the use of historical knowledge can be used to reduce ambiguities in requirements. With added knowledge, a second set of requirements was processed. Fewer requirements were found to be ambiguous. The results for this set were also a confirmation of the effectiveness of the proposed conceptual model.

The impact of these results caused an effect on how developers communicate and how developers and customers communicate. Novice developers are assumed to have no formal training in the area of Web-based form requirements. By using the proposed conceptual model and the assistance of an expert developer, a novice developer is able to understand and to reduce ambiguities in customer requirements, to reduce the time meeting with the customer, and learn about the entire process of reviewing and reducing ambiguities in customer requirements.

Limitations and Future Work

One important aspect of the study presented here was the use of a conceptual model and a scripted process for reducing ambiguities in customer requirements while improving communication among the people involved. The performance evaluation of the conceptual model and the process are reasonably hard to determine when both concepts must work together. In addition, the accuracy of the results of the proposed concept was highly dependable on the accuracy of the collected data and the involvement of the people collecting the data. For instance, the form employed in this study for the analysis of ambiguities in customer requirements was mostly biased. Although it was a relatively easy form, it depended on the perception of those who were using the form. The same could be applied to the measurement employed in this study for identifying ambiguities in requirements. The proposed study presented additional limitations and future work as described.

Scalability

The proposed model and process are limited to organizations with small budget and personnel. It can also be assumed that the organization would have no formal process in place for requirements elicitation and software development. The complete implementation of this theory in a university domain or a larger organization would be too big to put into practice and would require more study in this area. Also, as the results showed, presumably with all things, more data would be necessary to really show a difference in the results.

Context of Use

It is assumed that a novice developer has no formal training in specifying requirements and would require assistance from an expert developer when using the process proposed here in this study. The conceptual model is limited to improving customer requirements given a list of requirements. Other aspects of the requirement such as its functionality would require more elaboration and formalization of the model. The functionality of the requirement at the programming level could be included as part of the requirement sentence. The functionality field of the requirement sentence would be the ideal place for including the partial pseudocode for the functionality of the requirement, including the placement of the requirement on the form.

Additional Processing

The concept proposed here should not be the only method for reducing ambiguities in customer requirement. The proposed method should be used where suitable and with the support of other techniques for reducing ambiguities in customer requirements. It is essential to understand that the document produced after the use of the proposed concept and process must not be considered as the final requirements document. The document produced is just a draft version of a set of requirements with fewer ambiguities that can be used in subsequent iterations of the process and eventual customer approval.

User Interface and Files

Currently the user interface is limited to command line input and file processing. The implementation of a graphical user interface (GUI) can be put into practice in the future as more users are allowed to utilize the process. The handling of files for storing data is also archaic, and methods for storing data in a database can be evaluated for future improvement of the tool. The use of database is not well-suited when employing reasoning rules for processing data.

Additional Future Research

The extension of this study includes coordination between applications that are related in terms of common fields in the form. In the future, this study could be extended to complete the requirement elicitation process and possibly the entire software development process. This completion could be accomplished through the implementation of a formal process similar to the PSP (Personal Software Process) for software development.

Allowing customers to actually input the requirements into the tool is envisioned for the improvement of this conceptual model. Also plans are in place to also include the visual output of how the requirements will look on the form. Although the presented conceptual model currently does not allow customer to utilize the process when entering the requirements, as a future direction this can become possible with the use of modeling tools for visualizing the requirements. Customers often do not know all of the specifics of coding and building the code, but by adding a visual element to the conceptual model, it would be possible to produce a visual interpretation of the requirement.

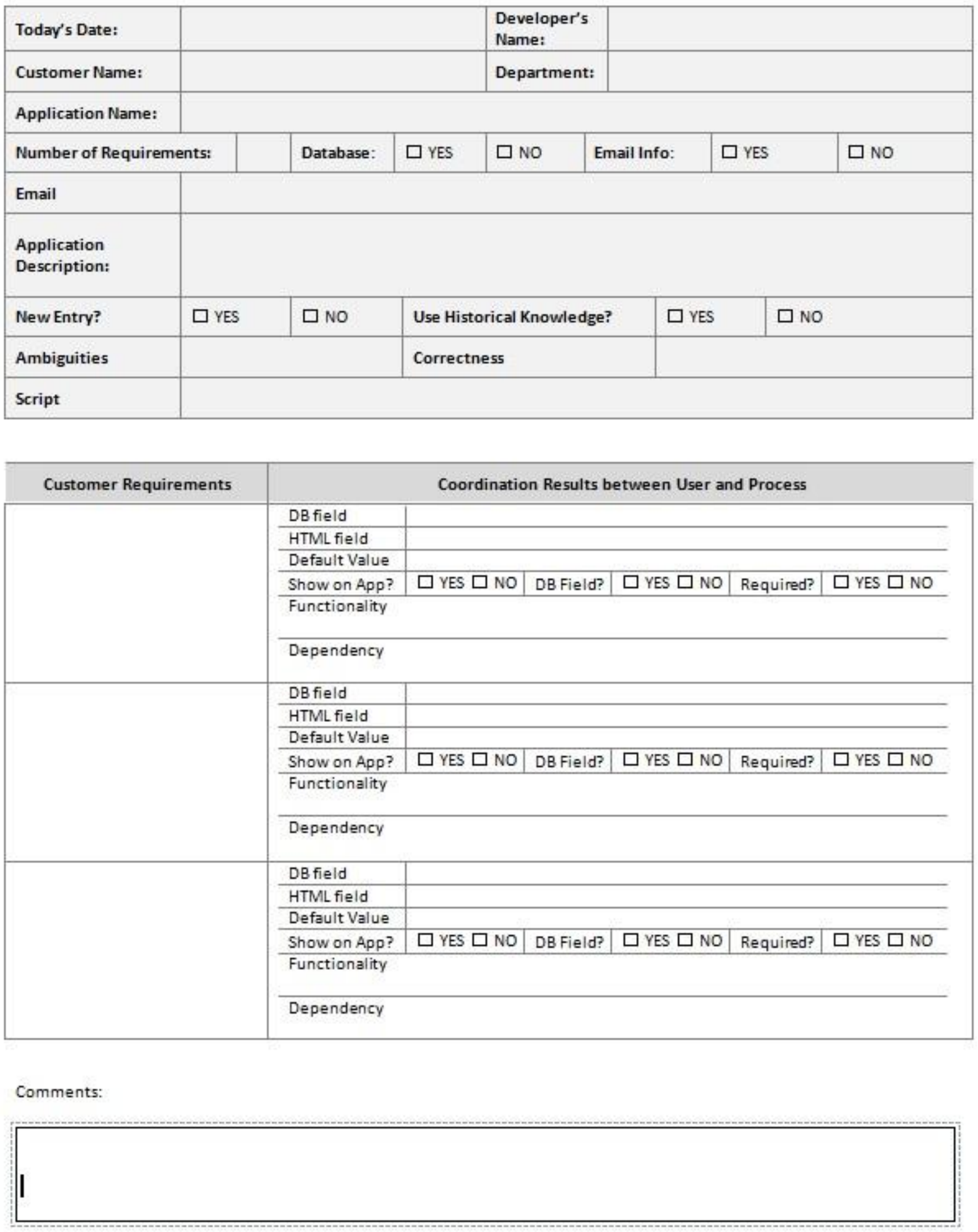
APPENDIX A

SCRIPTS, FORMS, AND INSTRUCTIONS

Process Script for Customer Requirements

Purpose:	To guide customers and developers in reducing ambiguities in requirements
Entry Criteria	- Identify customer name and department - Identify application name and details - Preliminary Customer Requirements Form - Requirement Processing Time Recording Log Form
Planning	- Customer(s) o Write requirements ▪ May meet with developer o Record customer requirements and application details in Preliminary Customer Requirements form o Input application details in Requirement Processing Time Recording Log form
Processing	- Customer o Process each customer requirement o Record process results for each requirement in Preliminary Customer Requirements Form o Record detailed time spent in Requirement Processing Time Recording Log form
Evaluation	- Developers o Analyze requirements ▪ Approximate unambiguity for requirements ▪ Prepare version 2 of Preliminary Customer Requirements Form o Meets with customer(s) to: ▪ Discuss results ▪ Discuss refinement • Refine customer requirements (cycle: Development and Testing) o Record detailed time spent in Requirement Processing Time Recording Log form
Exit Criteria	- Customer verify requirements o Meet with developer for review o Design and development may begin - Number of unambiguous requirements are recorded - Fewer ambiguities in customer requirements - A properly documented process for eliciting customer requirements and reducing ambiguities in customer requirements - A process that learns from history of previously defined requirements

Preliminary Customer Requirements Form



Today's Date:			Developer's Name:				
Customer Name:			Department:				
Application Name:							
Number of Requirements:		Database:	☐ YES	☐ NO	Email Info:	☐ YES	☐ NO
Email							
Application Description:							
New Entry?	☐ YES	☐ NO	Use Historical Knowledge?	☐ YES	☐ NO		
Ambiguities		Correctness					
Script							

Customer Requirements	Coordination Results between User and Process					
	DB field					
	HTML field					
	Default Value					
	Show on App?	☐ YES ☐ NO	DB Field?	☐ YES ☐ NO	Required?	☐ YES ☐ NO
	Functionality					
	Dependency					
	DB field					
	HTML field					
	Default Value					
	Show on App?	☐ YES ☐ NO	DB Field?	☐ YES ☐ NO	Required?	☐ YES ☐ NO
	Functionality					
	Dependency					
	DB field					
	HTML field					
	Default Value					
	Show on App?	☐ YES ☐ NO	DB Field?	☐ YES ☐ NO	Required?	☐ YES ☐ NO
	Functionality					
	Dependency					

Comments:

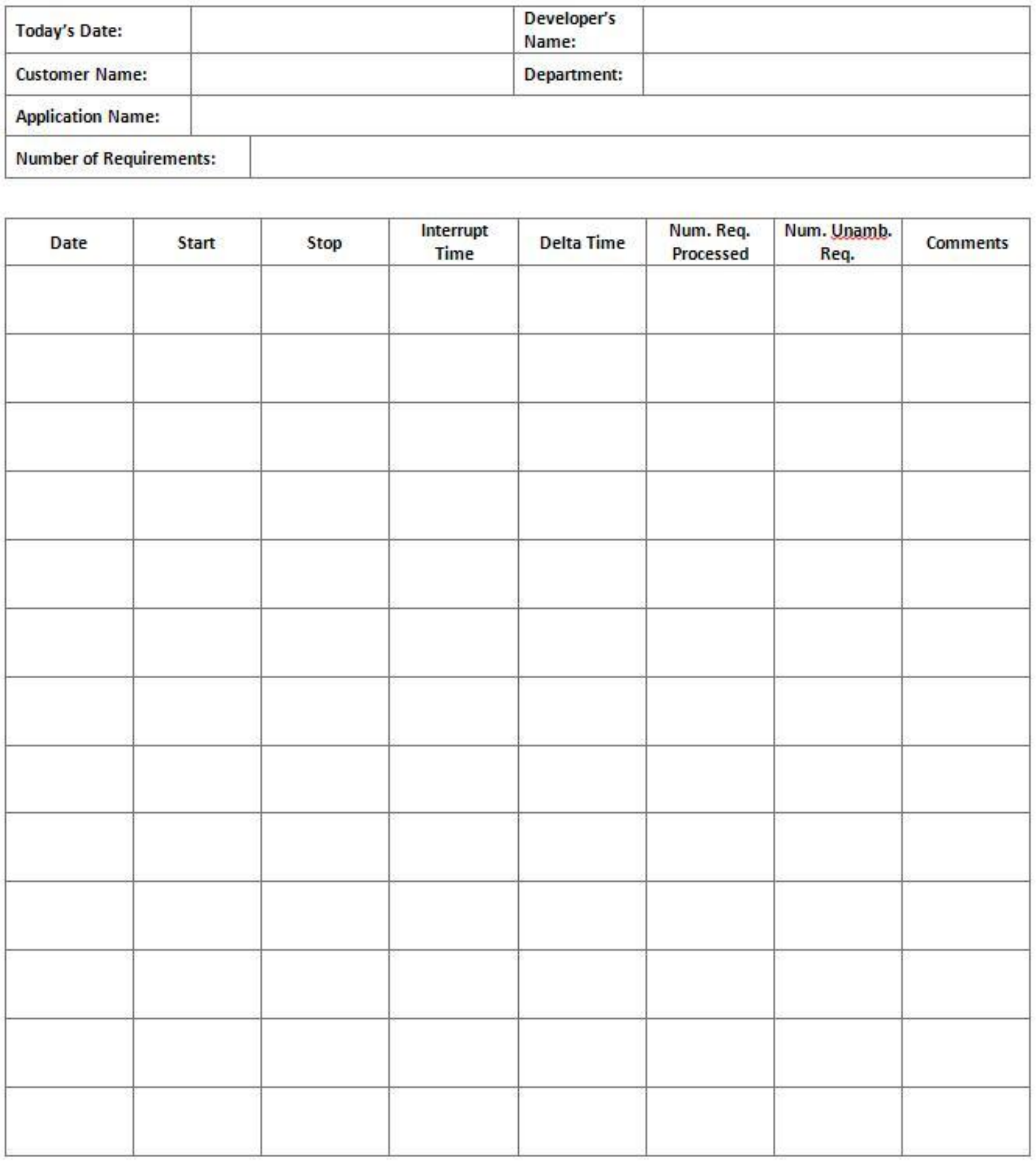
Preliminary Customer Requirements Form Instructions

Purpose	This form holds details about customer requirements for a given Web-based application
Header	Enter the following in the fields: - **Today's date** - Your name **(Developer's Name)** - **Customer Name** - Customer department (**Department**) - The name of the application for which these requirements will be employed (**Application Name**) - Total number of requirements in this form (**Number of Requirements**) - **Database**: mark "YES" if the application will be backed by a database to store its data or "NO" if the application will not be backed by a database - **Email Info**: mark "YES" if the application will be sending via email information entered in the application or "NO" if the application will not be sending information via email - If you marked "YES" in the Email Info box, enter the **Email** in which information entered in the application will be sent to - Enter the details of the application in the **Application Description** - **New Entry**: mark "YES" if the requirements will be entered in the tool with no prior knowledge involved or mark "NO" if not or if you are not sure - **Use Historical Knowledge**: mark "YES" if the you wish to load prior knowledge about other Web-based applications or mark "NO" if not of if you are not sure - **Ambiguities**: this field will be populated when you meet with the developer; this is calculated using the formula: *# {the requirements items that are mapped into concepts that can be traced from each other through relationships}/# {requirements items}* - **Correctness**: this field will be populated when customer meets with the developer; the value here represents the number of requirements that unambiguous. - **Script**: type here the process script you are using to fill this form. The script used may be one of the following: Process Script for Customer Requirements with No Historical Data or Process Script for Customer Requirements with Historical Data
Customer Requirements	In this column, enter the requirement for the application being developed. Enter as much detail as you know about the requirement. You may use a separate sheet for this step if the requirement has details that will not fit in the box.

(continued)

Coordination Results between User and Process	In this column, enter details about the results of each requirement upon using the process. - **DB field**: enter here the entire details about this item as per tool results - **HTML field**: enter here the entire details about this item as per tool results - **Default Value**: enter here the default value for this item. A default value is given to a requirement if no value is entered in the application - **Show on app?** Mark “YES” if the requirement will be shown on the application or mark “NO” if not or if you are not sure. - **DB Field**: Mark “YES” if the requirement will be a database item or mark “NO” if not or if you are not sure. - **Required**? Mark “YES” if the requirement is required on the form or mark “NO” if the requirement is not required on the form - **Functionality**: enter here the functionality about the requirement if any. For example, e-mail address must be in the format of local-part@domain. - **Dependency**: enter here the dependency criteria for the requirement. There are some requirements that are dependent on the values entered in the application. For instance, if you filled in a value for ACT (composite), then the SAT (composite) field is not required and vise-a-versa
Comments	Enter comments about requirements and any suggestions about the results from the tool coordination.

Requirement Processing Time Recording Log



Today's Date:		Developer's Name:	
Customer Name:		Department:	
Application Name:			
Number of Requirements:			

Date	Start	Stop	Interrupt Time	Delta Time	Num. Req. Processed	Num. Unamb. Req.	Comments

Requirement Processing Time Recording Log Instructions

Purpose	This form holds details about the time spent in processing customer requirements for a given Web-based application. Novice developer records all time spent in processing customer requirements from the time it was received to the time a requirement draft was produced.
Header	Enter the following in the fields: - **Today's date** - Your name (**Developer's Name)** - Customer name (**Customer Name**) - Customer department (**Department**) - The name of the application for which these requirements will be employed (**Application Name**) - Total number of requirements in this form (**Number of Requirements**)
Date	Enter the date when the process started, example: 11/20
Start	Enter the time when the process started, example: 9:30 am
Stop	Enter the time when you stop processing the requirements, example: 11:30 am
Interruption Time	Enter any interruption time that was not spent processing the requirements and the reason, for example: 1 hours, lunch break
Delta Time	Enter the actual time you spent processing the requirement minus the interruption time, for example 9:30 am to 2:00 pm, less 1 hour
Num. Req. Processed	Enter number of requirements processed during this time. For example: processed all 20 requirements, or processed 5 requirements
Num. Unamb. Requirements	During the process, enter the number of requirements that were unambiguous, for example: 5 requirements
Comments	Enter any other relevant comments related to the process of these requirements that might be useful later in case you have to come back to this same process
Important	If accurate time is not possible to be input here, enter the best estimate of the time. It is important to have all time spent processing the requirements recorded here.

APPENDIX B

CASE STUDIES REQUIREMENT SETS

Set of 27 requirements:

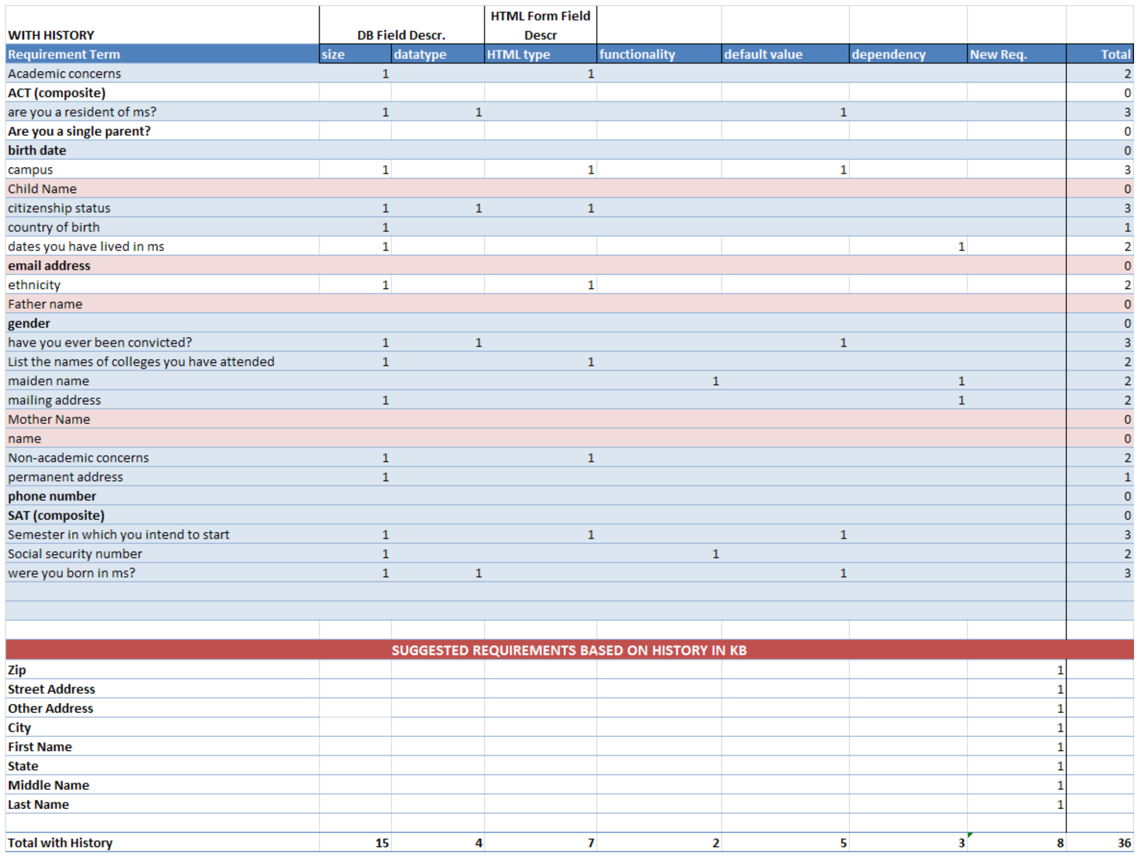
1. Social security number
2. Email address
3. Name
4. Maiden Name
5. Gender
6. Citizenship status
7. country of birth
8. Birth date
9. Ethnicity
10. Phone number
11. Permanent address
12. Mailing address
13. Campus
14. Are you a resident of Mississippi?
15. Were you born in Mississippi?
16. Dates you have lived in Mississippi?
17. Have you ever been convicted of a felony or do you currently have felony charges pending against you?
18. ACT (composite)
19. SAT (composite)
20. Semester in which you intend to start at Southern Miss
21. Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A.
22. Academic Concerns
23. Non-Academic Concerns
24. Are you a single parent?
25. Mother Name
26. Father Name
27. Child Name

Set of 15 requirements:

1. Ethnic Group
2. Are you a single parent?
3. If you are a student, do you receive Financial Aid?
4. Due Date
5. Zip
6. Student ID Number
7. Style Manual

8. Approximate Defense Date
9. Hometown (City and State)
10. Suffix
11. Current School OR School Last Attended
12. Date of Birth
13. Semester in which you intend to start at Southern Miss
14. Mailing address
15. Preferred First Name (if different from first name)

Result of the conceptual process for first set of requirements when historical knowledge is present.



WITH HISTORY	DB Field Descr.		HTML Form Field Descr					
Requirement Term	size	datatype	HTML type	functionality	default value	dependency	New Req.	Total
Academic concerns	1		1					2
ACT (composite)								0
are you a resident of ms?	1	1			1			3
Are you a single parent?								0
birth date								0
campus	1		1		1			3
Child Name								0
citizenship status	1	1	1					3
country of birth	1							1
dates you have lived in ms	1					1		2
email address								0
ethnicity	1		1					2
Father name								0
gender								0
have you ever been convicted?	1	1			1			3
List the names of colleges you have attended	1		1					2
maiden name				1		1		2
mailing address	1					1		2
Mother Name								0
name								0
Non-academic concerns	1		1					2
permanent address	1							1
phone number								0
SAT (composite)								0
Semester in which you intend to start	1		1		1			3
Social security number	1			1				2
were you born in ms?	1	1			1			3
SUGGESTED REQUIREMENTS BASED ON HISTORY IN KB								
Zip							1	
Street Address							1	
Other Address							1	
City							1	
First Name							1	
State							1	
Middle Name							1	
Last Name							1	
Total with History	**15**	**4**	**7**	**2**	**5**	**3**	**8**	**36**

Result of the conceptual process for second set of requirements when historical knowledge is present.

WITH HISTORY	DB Field Descr.		HTML Form Field Descr					
Requirement Term	**size**	**datatype**	**HTML type**	**functionality**	**default value**	**dependency**	**New Req.**	**Total**
Ethnic Group								0
Are you a single parent?								0
If you are a student, do you receive Financial Aid?	1	1	1	1	1			5
Due Date	1			1	1			3
Zip								0
Student ID Number								0
Style Manual								0
Approximate Defense Date	1			1	1			3
Hometown (City and State)								0
Suffix								0
Current School OR School Last Attended	1		1	1		1		4
Date of Birth								0
Semester in which you intend to start at Southern Miss								0
Mailing address								0
Preferred First Name (if different from first name)	1			1				2
Zip							1	
Street Address							1	
Other Address							1	
City							1	
First Name								
State							1	
Middle Name								
Last Name								
Total with History	**5**	**1**	**2**	**5**	**3**	**1**	**5**	**17**

APPENDIX C

PROPOSED PROCESS STEP-BY-STEP IMPLEMENTATION

In order to implement the proposed process as described in this dissertation, it is important to have completed the steps for setting up Eclipse and Jess described in Chapter IV of this dissertation. It is also important to have at hand the HTML code for the chosen Web-based forms and the equivalent “Create Table” SQL script for a specific department. If a form does not have a corresponding SQL “Create Table,” the HTML code is sufficient. However, if a form has multiple pages with multiple HTML pages, each page must have only one pair of <form> and </form> tag. The “name” attribute of the <form> tag must be the first attribute followed by “form”. It is recommended to place all HTML code between the <html> and </html> tags in a text document and saved with the .txt extension. In addition, the name for the text document must match the name of the SQL script for the Web-based form being parsed. As for the SQL “Create Table” script, it is recommend the file to remain with the .sql extension. The name of the Web-based application or department must be part of the SQL file name in addition to the name of the table. It is suggested to proceed with the implementation of this process using similar Web-based forms for a specific department.

The step-by-step instructions for implementing the proposed process for Web-based forms in a small organization are as follow:

1. A Java project in Eclipse named WebBasedFormProcess was created in the Workspace directory of Eclipse. In that project, the Java programs for parsing the HTML code and the SQL scripts were placed in the src folder. All Jess lines of code were placed in the WebBasedFormProcess folder.

2. A directory named HTML was created as a subdirectory of WebBasedFormProcess directory. Under the HTML directory, two subdirectories were created: FullHTML and ParsedHTML. Text file containing the HTML code for the chosen Web-based form were placed in the FullHTML folder.
3. A directory named SQL was created as a subdirectory of WebBasedFormProcess directory. Under the SQL directory, two subdirectories were created: FullSQL and ParsedSQL. The SQL “Create Table” script files for the chosen Web-based form were placed in the FullSQL folder.
4. The Java program “ParseHTMLFormFinal.java” was executed. This program parses the HTML code for the five chosen Web-based forms. This program requires the input directory where the files are located. The program processed one file at a time. The output of this program consists of a single file containing Jess “assert” statements for HTML form tags for all five forms. Figure A1 shows a simplified version of the contents of the input file this program processed. Figure A2 shows a simplified version of the contents of the generated output file.

```
        <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
        <html lang="en">
        <head>
        <title>Admissions |
        </title>
        </head>
        <body>
        <form name="page1" action="./index.php" method="post" id="page1">
        <input type="hidden" name="submitForm" value="page1">
        <table width="730" border="0" align="center" cellpadding="5" cellspacing="0"
bordercolor="#000000">
        <tbody>
        <tr><td align="left" valign="top" class="questionbox"><a
name="main_content"></a>
        <table width="100%" border="0" cellspacing="0" cellpadding="3">
        <tr><td colspan="3">Message about this application goes here.<br><br>
        <span style="font-style:italic;">All fields are required unless
specified.</span><br><br></td></tr>
        <tr><td width="25%" valign="middle"><div align="right"><label
for="firstname">First Name:</label></div></td>
        <td colspan="2" valign="middle"><input name=" firstname" type="text" id="
firstname" value="" size="30" maxlength="30" /></td></tr>
        ……….
        </tr></table></td>
        </tr></tbody></table>
        </form>
        </body>
        </html>
```

Figure A1: Simplified version of the HTML code for one of the chosen Web-based form.

```
        (deffunction process_HTMLInstance()
        (assert (Department (deptName "Admissions ")))
        (assert (Apps (hasDepartment "Admissions ")(appName
"gcscholarships")(hasSQLTables "gcscholarships_form")))

        (assert (Input(inApp "gcscholarships") (unique_id "firstname")
(input_name "firstname") (maxlength 128) (size 30) (is_of_type "text") (value
" ") (hasSQLObjects "firstname") (description "First Name")))
        (assert (MapObjects (objName "First Name") (hasHTMLObj
"Input")(hasSQLObjects "firstname") (isPartOf "gcscholarships")))
        ………
        )
```

Figure A2: Simplified version of the contents of the output file generated after the execution of the "ParseHTMLFormFinal.java" program.

Notice in Figure A1 all contents between the <form> and </form> pair of tags are parsed. The resulting output file must be placed in the ParsedHTML folder. The simplified version shown in Figure A2 shows the assert statements for each parsed field between the form tags. The contents of this file contain a Jess function in which the body contains the "assert" statements for creating instances of each field in the form. In addition, the department for this form and the name of the application are also shown. The name of the department is retrieved from the value between the <title> and </title>

tags. As a requirement, the name of the department must be placed between these tags followed by the | (bar) sign. The name of the application was retrieved from the name of the file. The name of the file must match the name of the application followed by an underscore, "_" and the name of the SQL table. In fact, the name of the HTML text file and the name of the SQL "Create Table" script file must match. The generated output file must go through a find and replace process. Everywhere in the contents of the file, a pair of double quotes without anything in between must be replaced with a single space between the double quotes. For instance, replace "" with " " in the generated HTML assert function for the HTML parsed code.

5. The Java program "ParseCreateTableFinal.java" was executed. This program parses the SQL "Create Table" scripts of the chosen HTML form. The input directory is chosen, and all SQL scripts located in the input directory are processed one at a time. Thus, a single output file is generated. The output file contains a Jess function with "assert" statements for the table and columns. Figure A3 contains a simplified version of the content of the input file required for parsing the "Create Table" script.

```
--------------------------------------------------------
--  File created - Tuesday-April-24-2012
--------------------------------------------------------
--------------------------------------------------------
--  DDL for Table FORM
--------------------------------------------------------

  CREATE TABLE "GCSCHOLARSHIP"."FORM"
   (     "FORMID" NUMBER(10,0),
         "FIRSTNAME" VARCHAR2(128 BYTE),
         "MIDDLENAME" VARCHAR2(128 BYTE) DEFAULT '',
         "LASTNAME" VARCHAR2(128 BYTE),
         "EMPLID" VARCHAR2(11 BYTE),
         "STREET" VARCHAR2(128 BYTE),
         "CITY" VARCHAR2(32 BYTE),
         "STATE" VARCHAR2(2 BYTE),
         "ZIP" VARCHAR2(10 BYTE),
         "DOB" VARCHAR2(11 BYTE),
         "PRIMARYPHONE" VARCHAR2(12 BYTE) DEFAULT ' ',
         "EMAIL" VARCHAR2(128 BYTE),
         "LASTSCHOOL" VARCHAR2(128 BYTE),
         "LASTDATEATTENDANCE" VARCHAR2(24 BYTE),
         "SCHOLARSHIPSEMESTERYEAR" VARCHAR2(24 BYTE) DEFAULT ' ',
         "ALREADYAPPLIED" CHAR(1 BYTE) DEFAULT 'N',
         "DATEAPPLIED" VARCHAR2(24 BYTE) DEFAULT ' ',
         "APPLICATIONDATE" TIMESTAMP (6),
         "IPADDRESS" VARCHAR2(40 BYTE),
"EXTRACTDATE" TIMESTAMP (6)
   )
……
```

Figure A3: Simplified version of the SQL “Create Table” script for one of the chosen Web-based form.

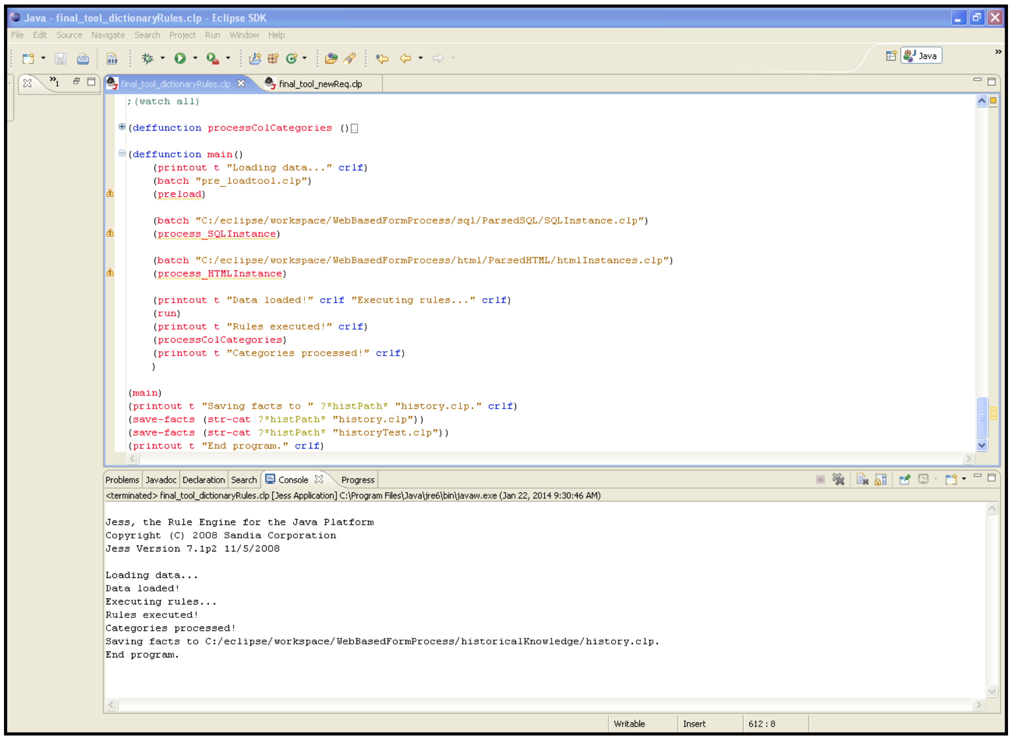
```
        (deffunction process_SQLInstance()

        (assert (SQLTable (table name "gcscholarships_form")))
        (assert (SQLColumns (colName "FORMID") (size 10) (colType "number") (tableName
"gcscholarships_form")))
        (assert (SQLColumns (colName "FIRSTNAME") (size 128) (colType "varchar2")
(tableName "gcscholarships_form")))
        (assert (SQLColumns (colName "MIDDLENAME") (size 128) (colType "varchar2")
(tableName "gcscholarships_form")))
        ……
        )
```

Figure A4: Simplified version of the contents of the output file generated after running the “ParseCreateTableFinal.java” program.

As seen in Figure A3, the contents in the SQL file for the “Create Table” script contains the details of a table creation and its corresponding columns. Each Web-based form that makes use of a database has a corresponding SQL file script. The contents of the resulting file as shown in the simplified content of Figure A4 include a function and “assert” statements for creating instances of a table and respective columns. The attributes defined for the columns in the SQL script are defined as slots of the

SQLColumns template in Jess. The produced output file is placed in the ParsedSQL folder under the SQL folder.

6. The Jess program “final_tool_dictionary_rules.clp” was executed. This program builds the knowledge-base about requirements for the chosen Web-based forms. The program requires the fully qualified path of the HTML and SQL output files specified in the “main” function of the program. It is important not to change the name of these files and their respective location. Figure A5 shows the console results and part of the program code after execution of the Jess program for creating the knowledge-base.

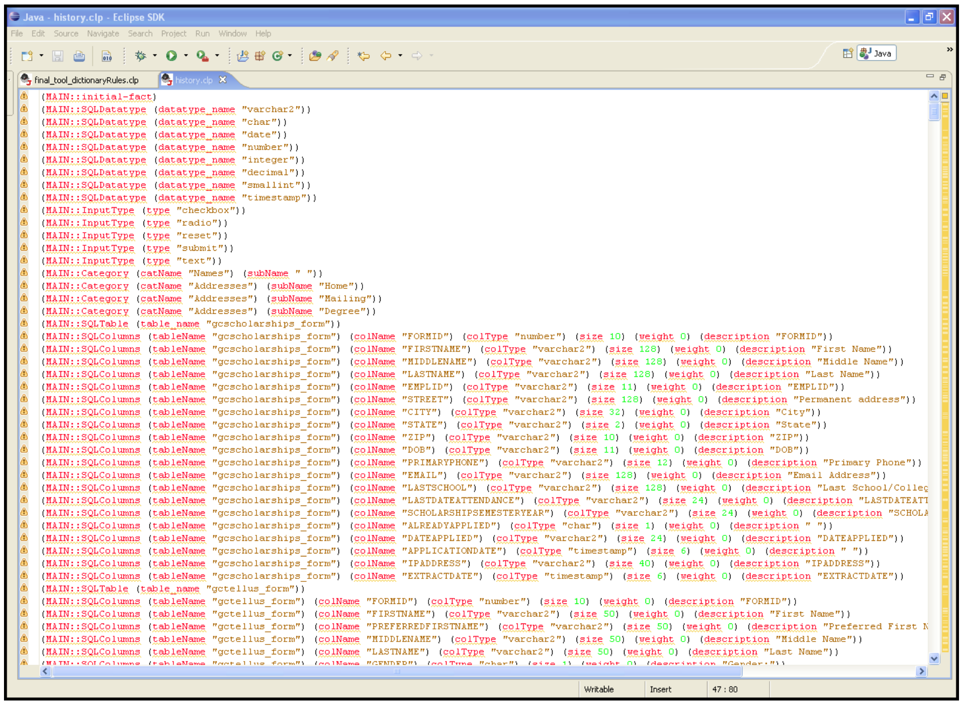

Java - final_tool_dictionaryRules.clp - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
final_tool_dictionaryRules.clp
final_tool_newReq.clp
Problems Javadoc Declaration Search Console Progress
Writable Insert 612 : 8

```
;(watch all)

(deffunction processColCategories ()

(deffunction main()
    (printout t "Loading data..." crlf)
    (batch "pre_loadtool.clp")
    (preload)

    (batch "C:/eclipse/workspace/WebBasedFormProcess/sql/ParsedSQL/SQLInstance.clp")
    (process_SQLInstance)

    (batch "C:/eclipse/workspace/WebBasedFormProcess/html/ParsedHTML/htmlInstances.clp")
    (process_HTMLInstance)

    (printout t "Data loaded!" crlf "Executing rules..." crlf)
    (run)
    (printout t "Rules executed!" crlf)
    (processColCategories)
    (printout t "Categories processed!" crlf)
   )

(main)
(printout t "Saving facts to " ?*histPath* "history.clp." crlf)
(save-facts (str-cat ?*histPath* "history.clp"))
(save-facts (str-cat ?*histPath* "historyTest.clp"))
(printout t "End program." crlf)
```

```
<terminated> final_tool_dictionaryRules.clp [Jess Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 22, 2014 9:30:46 AM)

Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008

Loading data...
Data loaded!
Executing rules...
Rules executed!
Categories processed!
Saving facts to C:/eclipse/workspace/WebBasedFormProcess/historicalKnowledge/history.clp.
End program.
```

Figure A5. Console result of the execution of the “final_tool_dictionary_rules.clp” program. The results shown do not show the output of the historical knowledge-base as the facts are saved to the history.clp file.

As seen in Figure A5, the program for creating the knowledge-base loads the files that were generated from the parsing of the HTML code, “htmlInstances.clp” and from the parsing of the SQL “Create Table” scripts, “SQLInstance.clp”. Once all “assert” statements from these files are executed, the rules associated to creating historical knowledge are also executed. Once all rules finish processing, a function for processing the categories is called and processed. All facts were saved into two different historical files for later usage. The contents of the historical file “history.clp” can be seen in Figure A6. The simplified version of this file shows the facts that were instantiated and saved. When historical knowledge is needed for processing new requirements, this file is utilized to load the facts into Jess’ main memory.


Java - history.clp - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
final_tool_dictionaryRules.clp
history.clp
Writable
Insert
47 : 80

```
(MAIN::initial-fact)
(MAIN::SQLDatatype (datatype_name "varchar2"))
(MAIN::SQLDatatype (datatype_name "char"))
(MAIN::SQLDatatype (datatype_name "date"))
(MAIN::SQLDatatype (datatype_name "number"))
(MAIN::SQLDatatype (datatype_name "integer"))
(MAIN::SQLDatatype (datatype_name "decimal"))
(MAIN::SQLDatatype (datatype_name "smallint"))
(MAIN::SQLDatatype (datatype_name "timestamp"))
(MAIN::InputType (type "checkbox"))
(MAIN::InputType (type "radio"))
(MAIN::InputType (type "reset"))
(MAIN::InputType (type "submit"))
(MAIN::InputType (type "text"))
(MAIN::Category (catName "Names") (subName " "))
(MAIN::Category (catName "Addresses") (subName "Home"))
(MAIN::Category (catName "Addresses") (subName "Mailing"))
(MAIN::Category (catName "Addresses") (subName "Degree"))
(MAIN::SQLTable (table_name "gcscholarships_form"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "FORMID") (colType "number") (size 10) (weight 0) (description "FORMID"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "FIRSTNAME") (colType "varchar2") (size 128) (weight 0) (description "First Name"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "MIDDLENAME") (colType "varchar2") (size 128) (weight 0) (description "Middle Name"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "LASTNAME") (colType "varchar2") (size 128) (weight 0) (description "Last Name"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "EMPLID") (colType "varchar2") (size 11) (weight 0) (description "EMPLID"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "STREET") (colType "varchar2") (size 128) (weight 0) (description "Permanent address"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "CITY") (colType "varchar2") (size 32) (weight 0) (description "City"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "STATE") (colType "varchar2") (size 2) (weight 0) (description "State"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "ZIP") (colType "varchar2") (size 10) (weight 0) (description "ZIP"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "DOB") (colType "varchar2") (size 11) (weight 0) (description "DOB"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "PRIMARYPHONE") (colType "varchar2") (size 12) (weight 0) (description "Primary Phone"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "EMAIL") (colType "varchar2") (size 128) (weight 0) (description "Email Address"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "LASTSCHOOL") (colType "varchar2") (size 128) (weight 0) (description "Last School/Colleg
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "LASTDATEATTENDANCE") (colType "varchar2") (size 24) (weight 0) (description "LASTDATEATT
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "SCHOLARSHIPSEMESTERYEAR") (colType "varchar2") (size 24) (weight 0) (description "SCHOLA
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "ALREADYAPPLIED") (colType "char") (size 1) (weight 0) (description " "))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "DATEAPPLIED") (colType "varchar2") (size 24) (weight 0) (description "DATEAPPLIED"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "APPLICATIONDATE") (colType "timestamp") (size 6) (weight 0) (description " "))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "IPADDRESS") (colType "varchar2") (size 40) (weight 0) (description "IPADDRESS"))
(MAIN::SQLColumns (tableName "gcscholarships_form") (colName "EXTRACTDATE") (colType "timestamp") (size 6) (weight 0) (description "EXTRACTDATE"))
(MAIN::SQLTable (table_name "gctellus_form"))
(MAIN::SQLColumns (tableName "gctellus_form") (colName "FORMID") (colType "number") (size 10) (weight 0) (description "FORMID"))
(MAIN::SQLColumns (tableName "gctellus_form") (colName "FIRSTNAME") (colType "varchar2") (size 50) (weight 0) (description "First Name"))
(MAIN::SQLColumns (tableName "gctellus_form") (colName "PREFERREDFIRSTNAME") (colType "varchar2") (size 50) (weight 0) (description "Preferred First N
(MAIN::SQLColumns (tableName "gctellus_form") (colName "MIDDLENAME") (colType "varchar2") (size 50) (weight 0) (description "Middle Name"))
(MAIN::SQLColumns (tableName "gctellus_form") (colName "LASTNAME") (colType "varchar2") (size 50) (weight 0) (description "Last Name"))
(MAIN::SQLColumns (tableName "gctellus_form") (colName "GENDER") (colType "char") (size 1) (weight 0) (description "Gender:"))
```

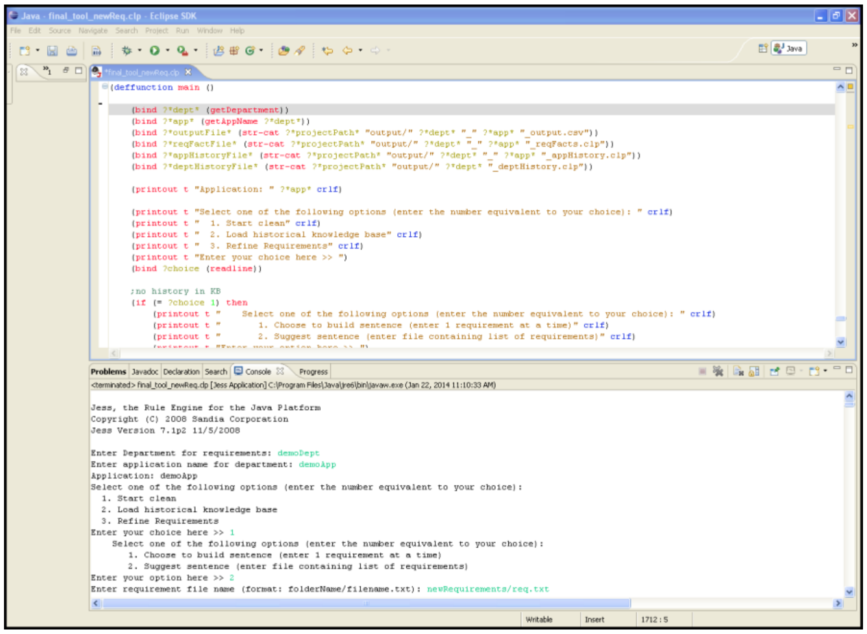
Figure A6. Simplified version of the history file storing all requirements derived from the chosen Web-based forms.

Each item in the chosen HTML forms and corresponding SQL “Create Table” script were processed and included in the historical files as facts. The next step was to process new requirements.

7. In this step, the “final_tool_newReq.clp” program was executed. This program allows for new requirements to be processed. The program allows the user to start a new historical knowledge-base for new requirements for a specific department without any prior knowledge about previously defined requirements or previous utilization of the historical knowledge (the historical facts produced in step 6) about previously defined requirements. The results of this program depend on the input requirements and the way the requirements are processed. If the process is carried out without any prior knowledge, the produced historical knowledge and draft requirement will not contain any prior knowledge about previously defined Web-based forms. While this process does not show much improvement in reducing ambiguities in the requirements, it does produce a suggested requirement sentence for each requirement. The lack of a complete requirement sentence is considered the main cause for ambiguity in Web-based form requirements. On the other hand, processing new requirements by making use of existing knowledge, new requirements, and existing requirements are suggested. The program produces requirement sentences for the new requirements that had a matching requirement in the knowledge-base including suggested requirements that fell in the same category.

Code execution without historical knowledge

Figure A7 shows the console of the results of running this program without any prior historical knowledge about the domain.

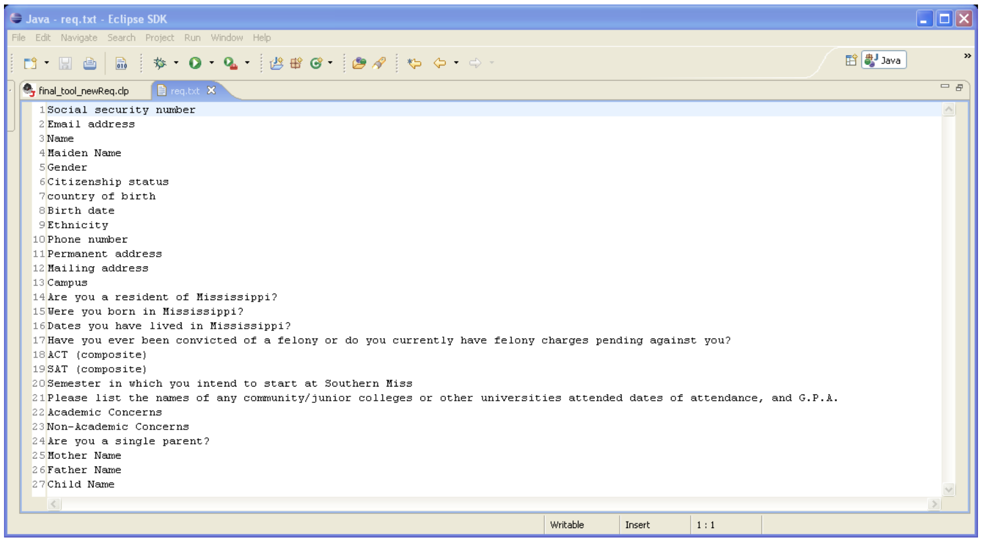


```
(deffunction main ()

    (bind ?*dept* (getDepartment))
    (bind ?*app* (getAppName ?*dept*))
    (bind ?*outputFile* (str-cat ?*projectPath* "output/" ?*dept* "_" ?*app* "_output.csv"))
    (bind ?*reqFactFile* (str-cat ?*projectPath* "output/" ?*dept* "_" ?*app* "_reqFacts.clp"))
    (bind ?*appHistoryFile* (str-cat ?*projectPath* "output/" ?*dept* "_" ?*app* "_appHistory.clp"))
    (bind ?*deptHistoryFile* (str-cat ?*projectPath* "output/" ?*dept* "_deptHistory.clp"))

    (printout t "Application: " ?*app* crlf)

    (printout t "Select one of the following options (enter the number equivalent to your choice): " crlf)
    (printout t "  1. Start clean" crlf)
    (printout t "  2. Load historical knowledge base" crlf)
    (printout t "  3. Refine Requirements" crlf)
    (printout t "Enter your choice here >> ")
    (bind ?choice (readline))

    ;no history in KB
    (if (= ?choice 1) then
        (printout t "    Select one of the following options (enter the number equivalent to your choice): " crlf)
        (printout t "       1. Choose to build sentence (enter 1 requirement at a time)" crlf)
        (printout t "       2. Suggest sentence (enter file containing list of requirements)" crlf)
```

```
<terminated> final_tool_newReq.clp [Jess Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 22, 2014 11:10:33 AM)

Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008

Enter Department for requirements: demoDept
Enter application name for department: demoApp
Application: demoApp
Select one of the following options (enter the number equivalent to your choice):
  1. Start clean
  2. Load historical knowledge base
  3. Refine Requirements
Enter your choice here >> 1
    Select one of the following options (enter the number equivalent to your choice):
       1. Choose to build sentence (enter 1 requirement at a time)
       2. Suggest sentence (enter file containing list of requirements)
Enter your option here >> 2
Enter requirement file name (format: folderName/filename.txt): newRequirements/req.txt
```

Figure A7. Simplified output of "final_tool_newReq.clp" program without the use of historical knowledge.

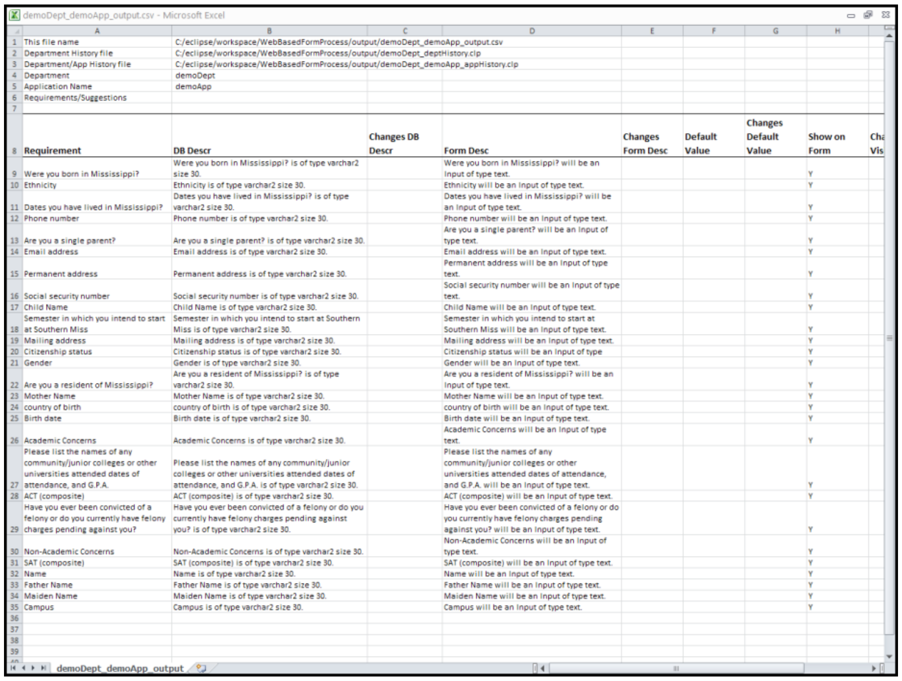
The input file "req.txt" processed in the execution of the program shown in Figure A7 contains the set of 27 requirements described in Chapter V of this dissertation. The file containing the set of 27 requirements is a text file. This file looks similar to the file shown in Figure A8.



```
Social security number
Email address
Name
Maiden Name
Gender
Citizenship status
country of birth
Birth date
Ethnicity
Phone number
Permanent address
Mailing address
Campus
Are you a resident of Mississippi?
Were you born in Mississippi?
Dates you have lived in Mississippi?
Have you ever been convicted of a felony or do you currently have felony charges pending against you?
ACT (composite)
SAT (composite)
Semester in which you intend to start at Southern Miss
Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A.
Academic Concerns
Non-Academic Concerns
Are you a single parent?
Mother Name
Father Name
Child Name
```

Figure A8. Set of 27 requirements for Web-based form processed.

After these requirements are processed, the suggested requirement sentences are written to a ".csv" file. Figure A9 shows the suggested requirement sentence file when viewed using Microsoft Office Excel spreadsheets applications. All SQL and HTML facts created are stored in the "history.clp" file if it exists or not.



demoDept_demoApp_output.csv - Microsoft Excel

	A	B	C	D	E	F	G	H	
1	This file name	C:/eclipse/workspace/WebBasedFormProcess/output/demoDept_demoApp_output.csv							
2	Department History file	C:/eclipse/workspace/WebBasedFormProcess/output/demoDept_deptHistory.clp							
3	Department/App History file	C:/eclipse/workspace/WebBasedFormProcess/output/demoDept_demoApp_appHistory.clp							
4	Department	demoDept							
5	Application Name	demoApp							
6	Requirements/Suggestions								
7									
8	**Requirement**	**DB Descr**	**Changes DB Descr**	**Form Desc**	**Changes Form Desc**	**Default Value**	**Changes Default Value**	**Show on Form**	**Cha Vis**
9	Were you born in Mississippi?	Were you born in Mississippi? is of type varchar2 size 30.		Were you born in Mississippi? will be an Input of type text.				Y	
10	Ethnicity	Ethnicity is of type varchar2 size 30.		Ethnicity will be an Input of type text.				Y	
11	Dates you have lived in Mississippi?	Dates you have lived in Mississippi? is of type varchar2 size 30.		Dates you have lived in Mississippi? will be an Input of type text.				Y	
12	Phone number	Phone number is of type varchar2 size 30.		Phone number will be an Input of type text.				Y	
13	Are you a single parent?	Are you a single parent? is of type varchar2 size 30.		Are you a single parent? will be an Input of type text.				Y	
14	Email address	Email address is of type varchar2 size 30.		Email address will be an Input of type text.				Y	
15	Permanent address	Permanent address is of type varchar2 size 30.		Permanent address will be an Input of type text.				Y	
16	Social security number	Social security number is of type varchar2 size 30.		Social security number will be an Input of type text.				Y	
17	Child Name	Child Name is of type varchar2 size 30.		Child Name will be an Input of type text.				Y	
18	Semester in which you intend to start at Southern Miss	Semester in which you intend to start at Southern Miss is of type varchar2 size 30.		Semester in which you intend to start at Southern Miss will be an Input of type text.				Y	
19	Mailing address	Mailing address is of type varchar2 size 30.		Mailing address will be an Input of type text.				Y	
20	Citizenship status	Citizenship status is of type varchar2 size 30.		Citizenship status will be an Input of type				Y	
21	Gender	Gender is of type varchar2 size 30.		Gender will be an Input of type text.				Y	
22	Are you a resident of Mississippi?	Are you a resident of Mississippi? is of type varchar2 size 30.		Are you a resident of Mississippi? will be an Input of type text.				Y	
23	Mother Name	Mother Name is of type varchar2 size 30.		Mother Name will be an Input of type text.				Y	
24	country of birth	country of birth is of type varchar2 size 30.		country of birth will be an Input of type text.				Y	
25	Birth date	Birth date is of type varchar2 size 30.		Birth date will be an Input of type text.				Y	
26	Academic Concerns	Academic Concerns is of type varchar2 size 30.		Academic Concerns will be an Input of type text.				Y	
27	Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A.	Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A. is of type varchar2 size 30.		Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A. will be an Input of type text.				Y	
28	ACT (composite)	ACT (composite) is of type varchar2 size 30.		ACT (composite) will be an Input of type text.				Y	
29	Have you ever been convicted of a felony or do you currently have felony charges pending against you?	Have you ever been convicted of a felony or do you currently have felony charges pending against you? is of type varchar2 size 30.		Have you ever been convicted of a felony or do you currently have felony charges pending against you? will be an Input of type text.				Y	
30	Non-Academic Concerns	Non-Academic Concerns is of type varchar2 size 30.		Non-Academic Concerns will be an Input of type text.				Y	
31	SAT (composite)	SAT (composite) is of type varchar2 size 30.		SAT (composite) will be an Input of type text.				Y	
32	Name	Name is of type varchar2 size 30.		Name will be an Input of type text.				Y	
33	Father Name	Father Name is of type varchar2 size 30.		Father Name will be an Input of type text.				Y	
34	Maiden Name	Maiden Name is of type varchar2 size 30.		Maiden Name will be an Input of type text.				Y	
35	Campus	Campus is of type varchar2 size 30.		Campus will be an Input of type text.				Y	
36									
37									
38									
39									

demoDept_demoApp_output

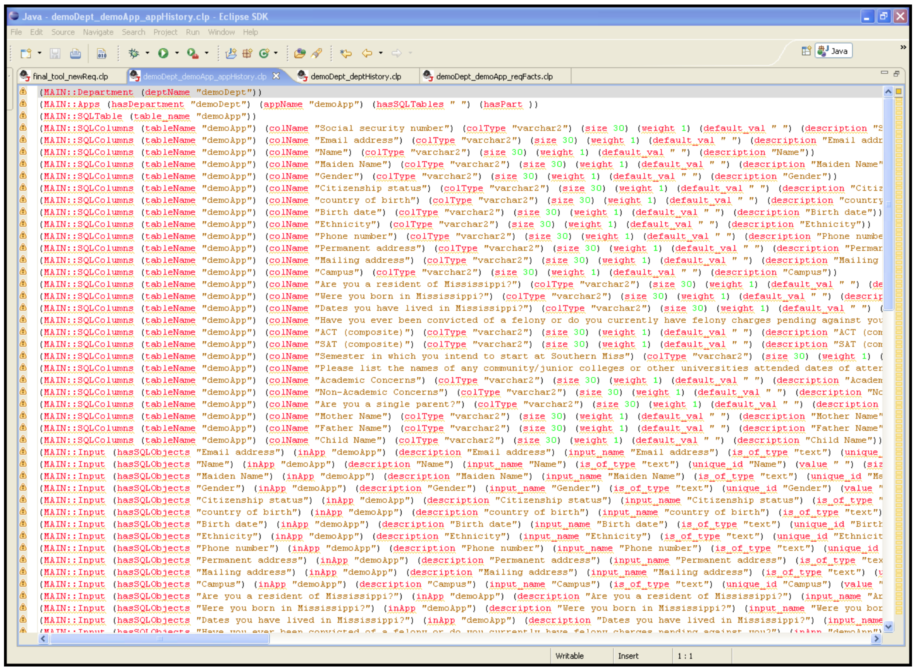
Figure A9. Suggested sentences for the set of 27 requirements processed without prior knowledge.

In addition to the ".csv" file, three other files are also generated as shown below:

1. "departmentName_appName_appHistory.clp" – as shown in Figure A10, this file stores SQL and HTML facts specific to the *departmentName* and *appName*.

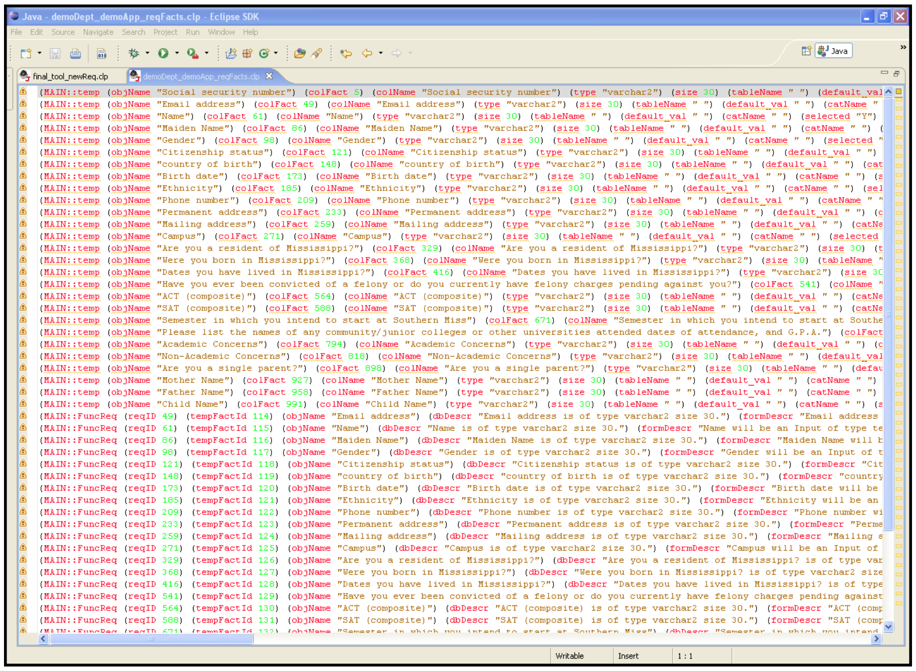
2. "departmentName_appName_reqFacts.clp" – as shown in Figure A11, this file stores the suggested requirements sentences as facts and temporary facts for the *departmentName* and *appName* to be later used during refinement.

3. “departmentName_deptHistory.clp” – as shown in Figure A12, this file stores SQL and HTML facts specific to the departmentName. All applications facts for the departmentName will be stored in this file.


Java - demoDept_demoApp_appHistory.clp - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
final_tool_newReq.clp
demoDept_demoApp_appHistory.clp
demoDept_deptHistory.clp
demoDept_demoApp_reqFacts.clp
Writable
Insert
1 : 1

```
(MAIN::Department (deptName "demoDept"))
(MAIN::Apps (hasDepartment "demoDept") (appName "demoApp") (hasSQLTables " ") (hasPart ))
(MAIN::SQLTable (table_name "demoApp"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Social security number") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "S
(MAIN::SQLColumns (tableName "demoApp") (colName "Email address") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Email addr
(MAIN::SQLColumns (tableName "demoApp") (colName "Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Name"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Maiden Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Maiden Name"
(MAIN::SQLColumns (tableName "demoApp") (colName "Gender") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Gender"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Citizenship status") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Citiz
(MAIN::SQLColumns (tableName "demoApp") (colName "country of birth") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "country
(MAIN::SQLColumns (tableName "demoApp") (colName "Birth date") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Birth date"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Ethnicity") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Ethnicity"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Phone number") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Phone numbe
(MAIN::SQLColumns (tableName "demoApp") (colName "Permanent address") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Perman
(MAIN::SQLColumns (tableName "demoApp") (colName "Mailing address") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Mailing
(MAIN::SQLColumns (tableName "demoApp") (colName "Campus") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Campus"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Are you a resident of Mississippi?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (de
(MAIN::SQLColumns (tableName "demoApp") (colName "Were you born in Mississippi?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (descrip
(MAIN::SQLColumns (tableName "demoApp") (colName "Dates you have lived in Mississippi?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (
(MAIN::SQLColumns (tableName "demoApp") (colName "Have you ever been convicted of a felony or do you currently have felony charges pending against you
(MAIN::SQLColumns (tableName "demoApp") (colName "ACT (composite)") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "ACT (com
(MAIN::SQLColumns (tableName "demoApp") (colName "SAT (composite)") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "SAT (com
(MAIN::SQLColumns (tableName "demoApp") (colName "Semester in which you intend to start at Southern Miss") (colType "varchar2") (size 30) (weight 1) (
(MAIN::SQLColumns (tableName "demoApp") (colName "Please list the names of any community/junior colleges or other universities attended dates of atter
(MAIN::SQLColumns (tableName "demoApp") (colName "Academic Concerns") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Academ
(MAIN::SQLColumns (tableName "demoApp") (colName "Non-Academic Concerns") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Nc
(MAIN::SQLColumns (tableName "demoApp") (colName "Are you a single parent?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description
(MAIN::SQLColumns (tableName "demoApp") (colName "Mother Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Mother Name"
(MAIN::SQLColumns (tableName "demoApp") (colName "Father Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Father Name"
(MAIN::SQLColumns (tableName "demoApp") (colName "Child Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Child Name"))
(MAIN::Input (hasSQLObjects "Email address") (inApp "demoApp") (description "Email address") (input_name "Email address") (is_of_type "text") (unique_
(MAIN::Input (hasSQLObjects "Name") (inApp "demoApp") (description "Name") (input_name "Name") (is_of_type "text") (unique_id "Name") (value " ") (siz
(MAIN::Input (hasSQLObjects "Maiden Name") (inApp "demoApp") (description "Maiden Name") (input_name "Maiden Name") (is_of_type "text") (unique_id "Ma
(MAIN::Input (hasSQLObjects "Gender") (inApp "demoApp") (description "Gender") (input_name "Gender") (is_of_type "text") (unique_id "Gender") (value "
(MAIN::Input (hasSQLObjects "Citizenship status") (inApp "demoApp") (description "Citizenship status") (input_name "Citizenship status") (is_of_type "
(MAIN::Input (hasSQLObjects "country of birth") (inApp "demoApp") (description "country of birth") (input_name "country of birth") (is_of_type "text")
(MAIN::Input (hasSQLObjects "Birth date") (inApp "demoApp") (description "Birth date") (input_name "Birth date") (is_of_type "text") (unique_id "Birth
(MAIN::Input (hasSQLObjects "Ethnicity") (inApp "demoApp") (description "Ethnicity") (input_name "Ethnicity") (is_of_type "text") (unique_id "Ethnicit
(MAIN::Input (hasSQLObjects "Phone number") (inApp "demoApp") (description "Phone number") (input_name "Phone number") (is_of_type "text") (unique_id
(MAIN::Input (hasSQLObjects "Permanent address") (inApp "demoApp") (description "Permanent address") (input_name "Permanent address") (is_of_type "tex
(MAIN::Input (hasSQLObjects "Mailing address") (inApp "demoApp") (description "Mailing address") (input_name "Mailing address") (is_of_type "text") (u
(MAIN::Input (hasSQLObjects "Campus") (inApp "demoApp") (description "Campus") (input_name "Campus") (is_of_type "text") (unique_id "Campus") (value "
(MAIN::Input (hasSQLObjects "Are you a resident of Mississippi?") (inApp "demoApp") (description "Are you a resident of Mississippi?") (input_name "Ar
(MAIN::Input (hasSQLObjects "Were you born in Mississippi?") (inApp "demoApp") (description "Were you born in Mississippi?") (input_name "Were you bor
(MAIN::Input (hasSQLObjects "Dates you have lived in Mississippi?") (inApp "demoApp") (description "Dates you have lived in Mississippi?") (input_name
(MAIN::Input (hasSQLObjects "Have you ever been convicted of a felony or do you currently have felony charges pending against you?") (inApp "demoApp")
```

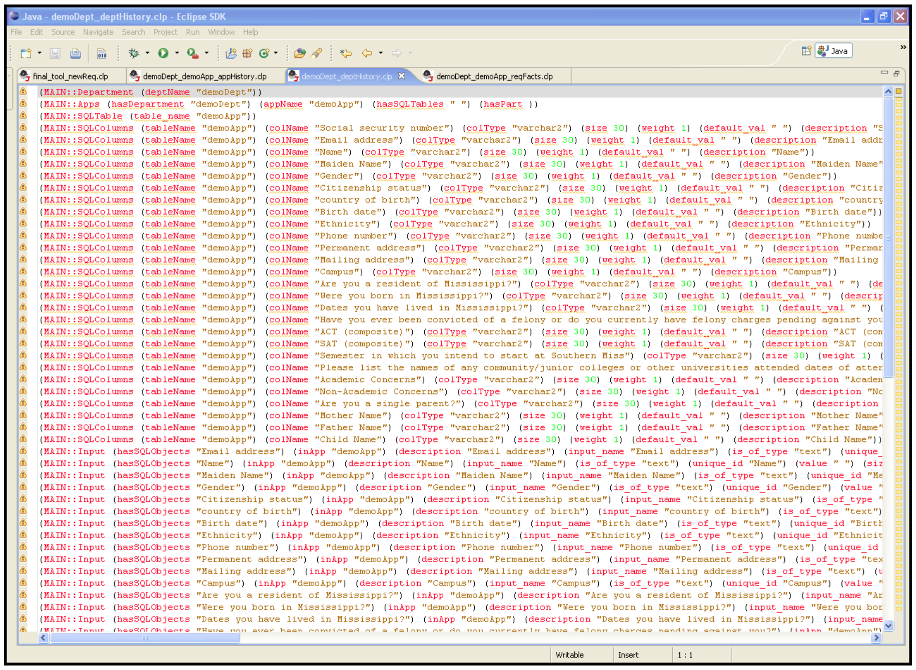
Figure A10. The contents of the “departmentName_appName_appHistory.clp” file.



Java - demoDept_demoApp_reqFacts.clp - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
final_tool_newReq.clp
demoDept_demoApp_reqFacts.clp
Writable
Insert
1 : 1

```
(MAIN::temp (objName "Social security number") (colFact 5) (colName "Social security number") (type "varchar2") (size 30) (tableName " ") (default_val
(MAIN::temp (objName "Email address") (colFact 49) (colName "Email address") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName "
(MAIN::temp (objName "Name") (colFact 61) (colName "Name") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ") (selected "Y")
(MAIN::temp (objName "Maiden Name") (colFact 86) (colName "Maiden Name") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ") (
(MAIN::temp (objName "Gender") (colFact 98) (colName "Gender") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ") (selected "
(MAIN::temp (objName "Citizenship status") (colFact 121) (colName "Citizenship status") (type "varchar2") (size 30) (tableName " ") (default_val " ")
(MAIN::temp (objName "country of birth") (colFact 148) (colName "country of birth") (type "varchar2") (size 30) (tableName " ") (default_val " ") (cat
(MAIN::temp (objName "Birth date") (colFact 173) (colName "Birth date") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ") (s
(MAIN::temp (objName "Ethnicity") (colFact 185) (colName "Ethnicity") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ") (sel
(MAIN::temp (objName "Phone number") (colFact 209) (colName "Phone number") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " "
(MAIN::temp (objName "Permanent address") (colFact 233) (colName "Permanent address") (type "varchar2") (size 30) (tableName " ") (default_val " ") (c
(MAIN::temp (objName "Mailing address") (colFact 259) (colName "Mailing address") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catNa
(MAIN::temp (objName "Campus") (colFact 271) (colName "Campus") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ") (selected
(MAIN::temp (objName "Are you a resident of Mississippi?") (colFact 329) (colName "Are you a resident of Mississippi?") (type "varchar2") (size 30) (t
(MAIN::temp (objName "Were you born in Mississippi?") (colFact 368) (colName "Were you born in Mississippi?") (type "varchar2") (size 30) (tableName "
(MAIN::temp (objName "Dates you have lived in Mississippi?") (colFact 416) (colName "Dates you have lived in Mississippi?") (type "varchar2") (size 3C
(MAIN::temp (objName "Have you ever been convicted of a felony or do you currently have felony charges pending against you?") (colFact 541) (colName "
(MAIN::temp (objName "ACT (composite)") (colFact 564) (colName "ACT (composite)") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catNa
(MAIN::temp (objName "SAT (composite)") (colFact 588) (colName "SAT (composite)") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catNa
(MAIN::temp (objName "Semester in which you intend to start at Southern Miss") (colFact 671) (colName "Semester in which you intend to start at Southe
(MAIN::temp (objName "Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A.") (colFact
(MAIN::temp (objName "Academic Concerns") (colFact 794) (colName "Academic Concerns") (type "varchar2") (size 30) (tableName " ") (default_val " ") (c
(MAIN::temp (objName "Non-Academic Concerns") (colFact 818) (colName "Non-Academic Concerns") (type "varchar2") (size 30) (tableName " ") (default_val
(MAIN::temp (objName "Are you a single parent?") (colFact 898) (colName "Are you a single parent?") (type "varchar2") (size 30) (tableName " ") (defau
(MAIN::temp (objName "Mother Name") (colFact 927) (colName "Mother Name") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ")
(MAIN::temp (objName "Father Name") (colFact 958) (colName "Father Name") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ")
(MAIN::temp (objName "Child Name") (colFact 991) (colName "Child Name") (type "varchar2") (size 30) (tableName " ") (default_val " ") (catName " ") (s
(MAIN::FuncReq (reqID 49) (tempFactId 114) (objName "Email address") (dbDescr "Email address is of type varchar2 size 30.") (formDescr "Email address
(MAIN::FuncReq (reqID 61) (tempFactId 115) (objName "Name") (dbDescr "Name is of type varchar2 size 30.") (formDescr "Name will be an Input of type te
(MAIN::FuncReq (reqID 86) (tempFactId 116) (objName "Maiden Name") (dbDescr "Maiden Name is of type varchar2 size 30.") (formDescr "Maiden Name will b
(MAIN::FuncReq (reqID 98) (tempFactId 117) (objName "Gender") (dbDescr "Gender is of type varchar2 size 30.") (formDescr "Gender will be an Input of t
(MAIN::FuncReq (reqID 121) (tempFactId 118) (objName "Citizenship status") (dbDescr "Citizenship status is of type varchar2 size 30.") (formDescr "Cit
(MAIN::FuncReq (reqID 148) (tempFactId 119) (objName "country of birth") (dbDescr "country of birth is of type varchar2 size 30.") (formDescr "country
(MAIN::FuncReq (reqID 173) (tempFactId 120) (objName "Birth date") (dbDescr "Birth date is of type varchar2 size 30.") (formDescr "Birth date will be
(MAIN::FuncReq (reqID 185) (tempFactId 121) (objName "Ethnicity") (dbDescr "Ethnicity is of type varchar2 size 30.") (formDescr "Ethnicity will be an
(MAIN::FuncReq (reqID 209) (tempFactId 122) (objName "Phone number") (dbDescr "Phone number is of type varchar2 size 30.") (formDescr "Phone number wi
(MAIN::FuncReq (reqID 233) (tempFactId 123) (objName "Permanent address") (dbDescr "Permanent address is of type varchar2 size 30.") (formDescr "Perma
(MAIN::FuncReq (reqID 259) (tempFactId 124) (objName "Mailing address") (dbDescr "Mailing address is of type varchar2 size 30.") (formDescr "Mailing a
(MAIN::FuncReq (reqID 271) (tempFactId 125) (objName "Campus") (dbDescr "Campus is of type varchar2 size 30.") (formDescr "Campus will be an Input of
(MAIN::FuncReq (reqID 329) (tempFactId 126) (objName "Are you a resident of Mississippi?") (dbDescr "Are you a resident of Mississippi? is of type var
(MAIN::FuncReq (reqID 368) (tempFactId 127) (objName "Were you born in Mississippi?") (dbDescr "Were you born in Mississippi? is of type varchar2 size
(MAIN::FuncReq (reqID 416) (tempFactId 128) (objName "Dates you have lived in Mississippi?") (dbDescr "Dates you have lived in Mississippi? is of type
(MAIN::FuncReq (reqID 541) (tempFactId 129) (objName "Have you ever been convicted of a felony or do you currently have felony charges pending against
(MAIN::FuncReq (reqID 564) (tempFactId 130) (objName "ACT (composite)") (dbDescr "ACT (composite) is of type varchar2 size 30.") (formDescr "ACT (comp
(MAIN::FuncReq (reqID 588) (tempFactId 131) (objName "SAT (composite)") (dbDescr "SAT (composite) is of type varchar2 size 30.") (formDescr "SAT (comp
(MAIN::FuncReq (reqID 671) (tempFactId 132) (objName "Semester in which you intend to start at Southern Miss") (dbDescr "Semester in which you intend
```

Figure A11. The contents of the “departmentName_appName_reqFacts.clp” file.



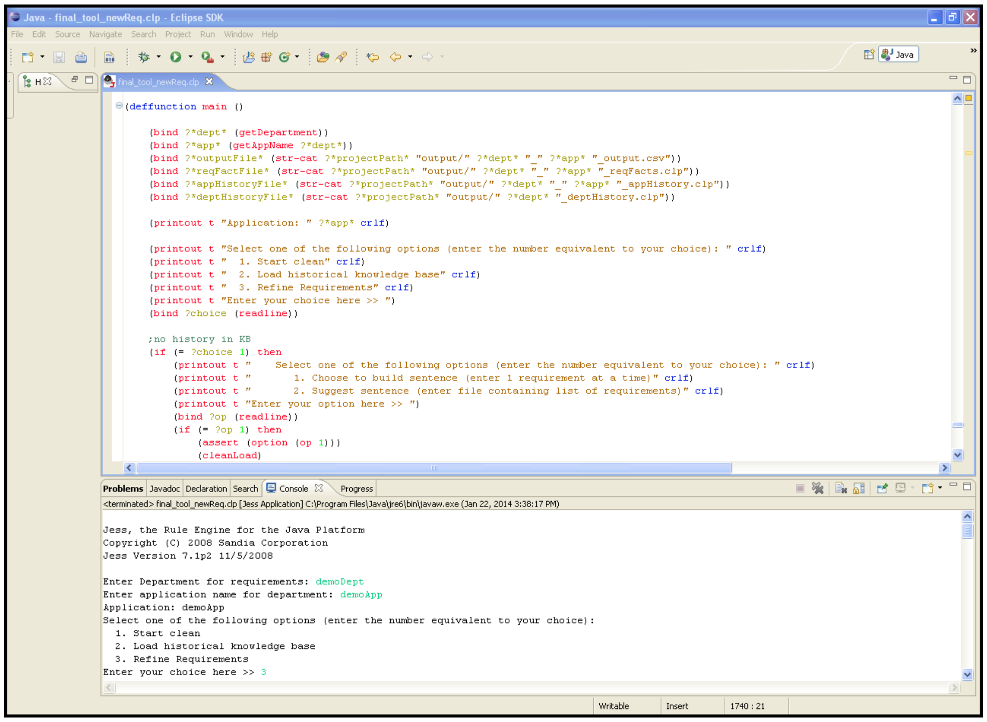
Java - demoDept_deptHistory.clp - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
final_tool_newReq.clp
demoDept_demoApp_appHistory.clp
demoDept_deptHistory.clp
demoDept_demoApp_reqFacts.clp
Writable
Insert
1 : 1

```
(MAIN::Department (deptName "demoDept"))
(MAIN::Apps (hasDepartment "demoDept") (appName "demoApp") (hasSQLTables " ") (hasPart ))
(MAIN::SQLTable (table_name "demoApp"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Social security number") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "S
(MAIN::SQLColumns (tableName "demoApp") (colName "Email address") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Email addr
(MAIN::SQLColumns (tableName "demoApp") (colName "Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Name"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Maiden Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Maiden Name"
(MAIN::SQLColumns (tableName "demoApp") (colName "Gender") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Gender"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Citizenship status") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Citiz
(MAIN::SQLColumns (tableName "demoApp") (colName "country of birth") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "country
(MAIN::SQLColumns (tableName "demoApp") (colName "Birth date") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Birth date"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Ethnicity") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Ethnicity"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Phone number") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Phone numbe
(MAIN::SQLColumns (tableName "demoApp") (colName "Permanent address") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Perman
(MAIN::SQLColumns (tableName "demoApp") (colName "Mailing address") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Mailing
(MAIN::SQLColumns (tableName "demoApp") (colName "Campus") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Campus"))
(MAIN::SQLColumns (tableName "demoApp") (colName "Are you a resident of Mississippi?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (de
(MAIN::SQLColumns (tableName "demoApp") (colName "Were you born in Mississippi?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (descrip
(MAIN::SQLColumns (tableName "demoApp") (colName "Dates you have lived in Mississippi?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (
(MAIN::SQLColumns (tableName "demoApp") (colName "Have you ever been convicted of a felony or do you currently have felony charges pending against you
(MAIN::SQLColumns (tableName "demoApp") (colName "ACT (composite)") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "ACT (com
(MAIN::SQLColumns (tableName "demoApp") (colName "SAT (composite)") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "SAT (com
(MAIN::SQLColumns (tableName "demoApp") (colName "Semester in which you intend to start at Southern Miss") (colType "varchar2") (size 30) (weight 1) (
(MAIN::SQLColumns (tableName "demoApp") (colName "Please list the names of any community/junior colleges or other universities attended dates of atter
(MAIN::SQLColumns (tableName "demoApp") (colName "Academic Concerns") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Academ
(MAIN::SQLColumns (tableName "demoApp") (colName "Non-Academic Concerns") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "No
(MAIN::SQLColumns (tableName "demoApp") (colName "Are you a single parent?") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description
(MAIN::SQLColumns (tableName "demoApp") (colName "Mother Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Mother Name"
(MAIN::SQLColumns (tableName "demoApp") (colName "Father Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Father Name"
(MAIN::SQLColumns (tableName "demoApp") (colName "Child Name") (colType "varchar2") (size 30) (weight 1) (default_val " ") (description "Child Name"))
(MAIN::Input (hasSQLObjects "Email address") (inApp "demoApp") (description "Email address") (input_name "Email address") (is_of_type "text") (unique_
(MAIN::Input (hasSQLObjects "Name") (inApp "demoApp") (description "Name") (input_name "Name") (is_of_type "text") (unique_id "Name") (value " ") (siz
(MAIN::Input (hasSQLObjects "Maiden Name") (inApp "demoApp") (description "Maiden Name") (input_name "Maiden Name") (is_of_type "text") (unique_id "Ma
(MAIN::Input (hasSQLObjects "Gender") (inApp "demoApp") (description "Gender") (input_name "Gender") (is_of_type "text") (unique_id "Gender") (value "
(MAIN::Input (hasSQLObjects "Citizenship status") (inApp "demoApp") (description "Citizenship status") (input_name "Citizenship status") (is_of_type "
(MAIN::Input (hasSQLObjects "country of birth") (inApp "demoApp") (description "country of birth") (input_name "country of birth") (is_of_type "text")
(MAIN::Input (hasSQLObjects "Birth date") (inApp "demoApp") (description "Birth date") (input_name "Birth date") (is_of_type "text") (unique_id "Birth
(MAIN::Input (hasSQLObjects "Ethnicity") (inApp "demoApp") (description "Ethnicity") (input_name "Ethnicity") (is_of_type "text") (unique_id "Ethnicit
(MAIN::Input (hasSQLObjects "Phone number") (inApp "demoApp") (description "Phone number") (input_name "Phone number") (is_of_type "text") (unique_id
(MAIN::Input (hasSQLObjects "Permanent address") (inApp "demoApp") (description "Permanent address") (input_name "Permanent address") (is_of_type "tex
(MAIN::Input (hasSQLObjects "Mailing address") (inApp "demoApp") (description "Mailing address") (input_name "Mailing address") (is_of_type "text") (u
(MAIN::Input (hasSQLObjects "Campus") (inApp "demoApp") (description "Campus") (input_name "Campus") (is_of_type "text") (unique_id "Campus") (value "
(MAIN::Input (hasSQLObjects "Are you a resident of Mississippi?") (inApp "demoApp") (description "Are you a resident of Mississippi?") (input_name "Ar
(MAIN::Input (hasSQLObjects "Were you born in Mississippi?") (inApp "demoApp") (description "Were you born in Mississippi?") (input_name "Were you bor
(MAIN::Input (hasSQLObjects "Dates you have lived in Mississippi?") (inApp "demoApp") (description "Dates you have lived in Mississippi?") (input_name
(MAIN::Input (hasSQLObjects "Have you ever been convicted of a felony or do you currently have felony charges pending against you?") (inApp "demoApp")
```

Figure A12. The contents of the “departmentName_deptHistory.clp” file.

Novice and expert developers go through each requirement shown in Figure A9. For each requirement that needs a change, the change is recorded in the corresponding column for that change and refined through the same program, the "final_tool_newReq.clp" program. After the requirements are processed, the requirements are then input into the Preliminary Customer Requirement Form to present to the customer for review and approval. Any changes to any one of the requirements after customer review, go through the refinement process again until an acceptable set of requirements is approved.

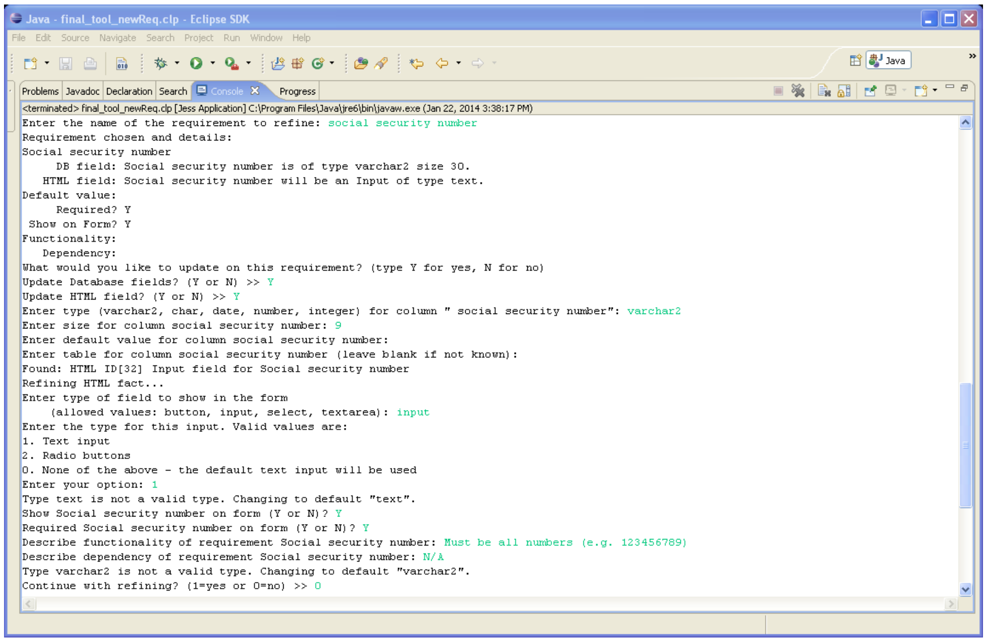
During the refinement process, the name of the department and the application must be known. The novice developer inputs this information and chooses option 3 to refine the requirement(s). As shown in Figures A13 and A14, each requirement needed to be refined is input and processed.


Java - final_tool_newReq.clp - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Java
final_tool_newReq.clp
Problems Javadoc Declaration Search Console Progress
Writable
Insert
1740 : 21

```
(deffunction main ()

    (bind ?*dept* (getDepartment))
    (bind ?*app* (getAppName ?*dept*))
    (bind ?*outputFile* (str-cat ?*projectPath* "output/" ?*dept* "_" ?*app* "_output.csv"))
    (bind ?*reqFactFile* (str-cat ?*projectPath* "output/" ?*dept* "_" ?*app* "_reqFacts.clp"))
    (bind ?*appHistoryFile* (str-cat ?*projectPath* "output/" ?*dept* "_" ?*app* "_appHistory.clp"))
    (bind ?*deptHistoryFile* (str-cat ?*projectPath* "output/" ?*dept* "_deptHistory.clp"))

    (printout t "Application: " ?*app* crlf)

    (printout t "Select one of the following options (enter the number equivalent to your choice): " crlf)
    (printout t "  1. Start clean" crlf)
    (printout t "  2. Load historical knowledge base" crlf)
    (printout t "  3. Refine Requirements" crlf)
    (printout t "Enter your choice here >> ")
    (bind ?choice (readline))

    ;no history in KB
    (if (= ?choice 1) then
        (printout t "    Select one of the following options (enter the number equivalent to your choice): " crlf)
        (printout t "       1. Choose to build sentence (enter 1 requirement at a time)" crlf)
        (printout t "       2. Suggest sentence (enter file containing list of requirements)" crlf)
        (printout t "Enter your option here >> ")
        (bind ?op (readline))
        (if (= ?op 1) then
            (assert (option (op 1)))
            (cleanLoad)
```

```
<terminated> final_tool_newReq.clp [Jess Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 22, 2014 3:38:17 PM)

Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008

Enter Department for requirements: demoDept
Enter application name for department: demoApp
Application: demoApp
Select one of the following options (enter the number equivalent to your choice):
  1. Start clean
  2. Load historical knowledge base
  3. Refine Requirements
Enter your choice here >> 3
```

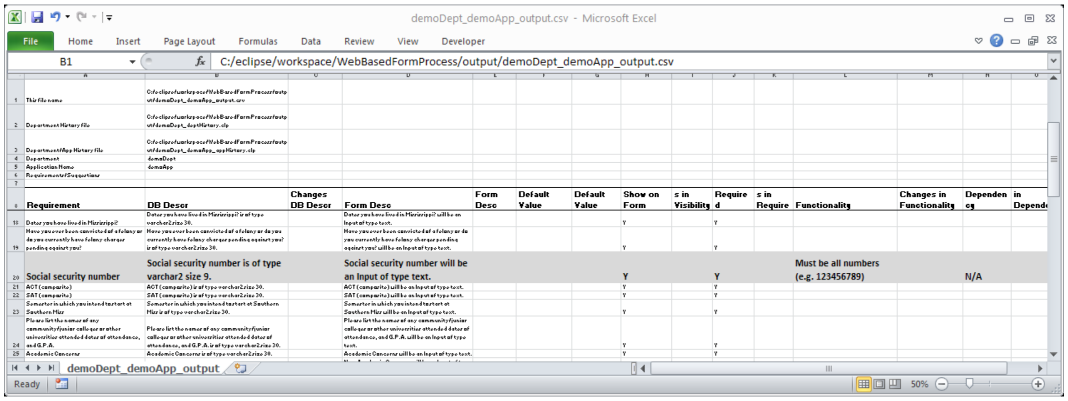
Figure A13: Console results when refining a requirement for a given department and application.


Java - final_tool_newReq.clp - Eclipse SDK
File Edit Source Navigate Search Project Run Window Help
Problems Javadoc Declaration Search Console Progress
Java

```
<terminated> final_tool_newReq.clp [Jess Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 22, 2014 3:38:17 PM)
Enter the name of the requirement to refine: social security number
Requirement chosen and details:
Social security number
     DB field: Social security number is of type varchar2 size 30.
   HTML field: Social security number will be an Input of type text.
Default value:
     Required? Y
 Show on Form? Y
Functionality:
   Dependency:
What would you like to update on this requirement? (type Y for yes, N for no)
Update Database fields? (Y or N) >> Y
Update HTML field? (Y or N) >> Y
Enter type (varchar2, char, date, number, integer) for column " social security number": varchar2
Enter size for column social security number: 9
Enter default value for column social security number:
Enter table for column social security number (leave blank if not known):
Found: HTML ID[32] Input field for Social security number
Refining HTML fact...
Enter type of field to show in the form
    (allowed values: button, input, select, textarea): input
Enter the type for this input. Valid values are:
1. Text input
2. Radio buttons
0. None of the above - the default text input will be used
Enter your option: 1
Type text is not a valid type. Changing to default "text".
Show Social security number on form (Y or N)? Y
Required Social security number on form (Y or N)? Y
Describe functionality of requirement Social security number: Must be all numbers (e.g. 123456789)
Describe dependency of requirement Social security number: N/A
Type varchar2 is not a valid type. Changing to default "varchar2".
Continue with refining? (1=yes or 0=no) >> 0
```

Figure A14. Console results when refining “social security number” requirement.

When refining a requirement, the currently defined SQL column fact and HTML fact must be edited or deleted in order for the creation of a new requirement fact. For instance, as shown in Figures A13 and A14, novice developer chooses to refine the “social security number” requirement. The requirement is found and the developer is prompted to enter the new information about the requirement being refined for both the SQL column fact and HTML fact. Once the values for the corresponding slot facts are entered, a new requirement fact is created and added to the corresponding fact list and files. As seen in Figure A15, the new “.csv” file containing the suggested sentences is generated which includes the newly created “social security number” requirement with its new definition and values.


demoDept_demoApp_output.csv - Microsoft Excel
File
Home
Insert
Page Layout
Formulas
Data
Review
View
Developer
B1
C:/eclipse/workspace/WebBasedFormProcess/output/demoDept_demoApp_output.csv
demoDept_demoApp_output
Ready
50%

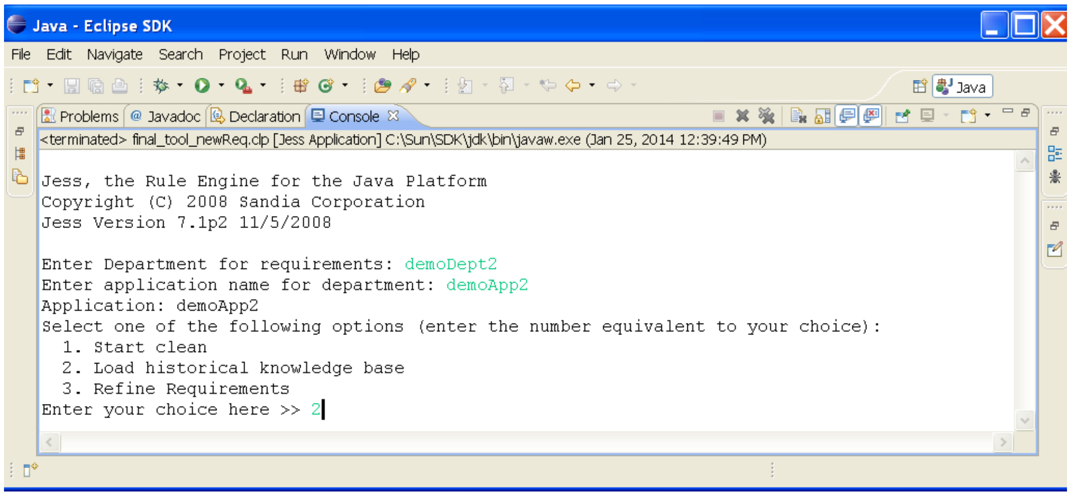
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	This file name	C:/eclipse/workspace/WebBasedFormProcess/output/demoDept_demoApp_output.csv													
2	Department History file	C:/eclipse/workspace/WebBasedFormProcess/output/demoDept_deptHistory.clp													
3	Department/App History file	C:/eclipse/workspace/WebBasedFormProcess/output/demoDept_demoApp_appHistory.clp													
4	Department	demoDept													
5	Application Name	demoApp													
6	Requirements/Suggestions														
7															
8	Requirement	DB Descr	Changes DB Descr	Form Desc	Form Desc	Default Value	Default Value	Show on Form	s in Visibility	Required	s in Require	Functionality	Changes in Functionality	Dependency	in Depende
18	Dates you have lived in Mississippi?	Dates you have lived in Mississippi? is of type varchar2 size 30.		Dates you have lived in Mississippi? will be an Input of type text.				Y		Y					
19	Have you ever been convicted of a felony or do you currently have felony charges pending against you?	Have you ever been convicted of a felony or do you currently have felony charges pending against you? is of type varchar2 size 30.		Have you ever been convicted of a felony or do you currently have felony charges pending against you? will be an Input of type text.				Y		Y					
20	Social security number	Social security number is of type varchar2 size 9.		Social security number will be an Input of type text.				Y		Y		Must be all numbers (e.g. 123456789)		N/A	
21	ACT (composite)	ACT (composite) is of type varchar2 size 30.		ACT (composite) will be an Input of type text.				Y		Y					
22	SAT (composite)	SAT (composite) is of type varchar2 size 30.		SAT (composite) will be an Input of type text.				Y		Y					
23	Semester in which you intend to start at Southern Miss	Semester in which you intend to start at Southern Miss is of type varchar2 size 30.		Semester in which you intend to start at Southern Miss will be an Input of type text.				Y		Y					
24	Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A.	Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A. is of type varchar2 size 30.		Please list the names of any community/junior colleges or other universities attended dates of attendance, and G.P.A. will be an Input of type text.				Y		Y					
25	Academic Concerns	Academic Concerns is of type varchar2 size 30.		Academic Concerns will be an Input of type text.				Y		Y					

Figure A15. Newly created ".csv" file containing the new definition for the "social security number" requirement.

If more than one requirement is refined, the ".csv" file shown in Figure A15 will contain the list of all requirements that were refined and those that were not. The file is generated after all requirements are refined, and the execution of the program is ended.

Code execution with historical knowledge

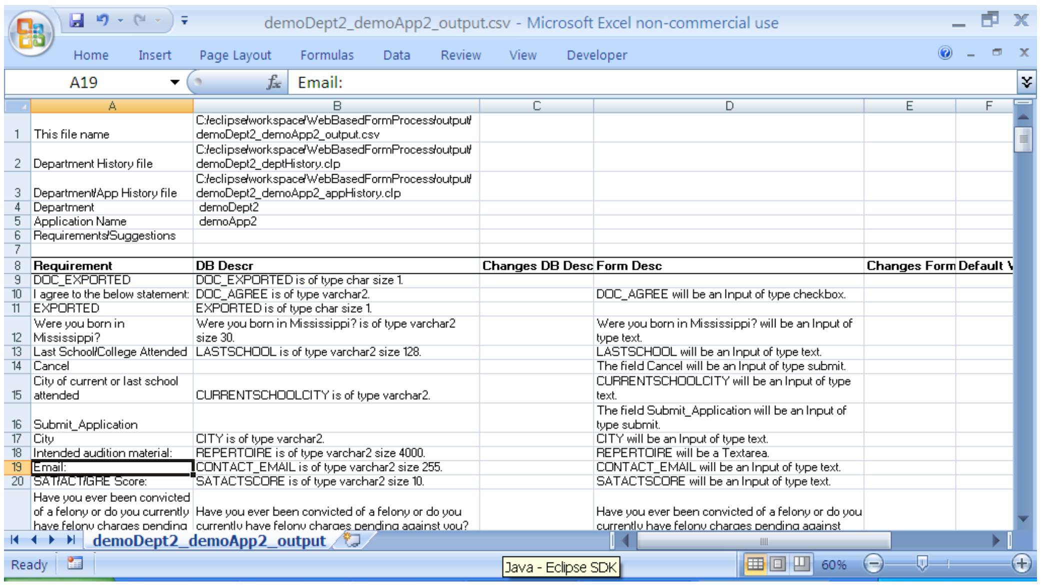
Executing the code using historical knowledge is basically the same process as when no historical knowledge is present, except in this process there will be history about previously defined requirements. The same set of 27 requirements will be input into the program and matched against existing requirements. Figure A16 shows the simplified output when historical knowledge is chosen for processing new requirements.


Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Console

```
<terminated> final_tool_newReq.clp [Jess Application] C:\Sun\SDK\jdk\bin\javaw.exe (Jan 25, 2014 12:39:49 PM)

Jess, the Rule Engine for the Java Platform
Copyright (C) 2008 Sandia Corporation
Jess Version 7.1p2 11/5/2008

Enter Department for requirements: demoDept2
Enter application name for department: demoApp2
Application: demoApp2
Select one of the following options (enter the number equivalent to your choice):
  1. Start clean
  2. Load historical knowledge base
  3. Refine Requirements
Enter your choice here >> 2
```

Figure A16. Simplified console result when executing requirements using historical data about previously defined requirements.

After the requirements are processed, the generated new and existing facts are saved to the history file and to the corresponding files as previously explained in step 7 when requirements are processed with no historical knowledge. If the requirements cannot be matched, a default requirement sentence is produced the same way as when no historical knowledge is available. However, when processing requirements using historical knowledge new requirements are also matched against requirements in a category. As explained in Chapter III and IV of this dissertation, categories are created to combine requirements that are often used in conjunction. Therefore, if a requirement does not have an exact match, requirements from a category are suggested. Figure A17 shows the simplified ".csv" file containing the suggested requirement sentences when historical knowledge is available.


demoDept2_demoApp2_output.csv - Microsoft Excel non-commercial use
Home
Insert
Page Layout
Formulas
Data
Review
View
Developer
A19
Email:
demoDept2_demoApp2_output
Ready
Java - Eclipse SDK
60%

	A	B	C	D	E	F
1	This file name	C:/eclipse/workspace/WebBasedFormProcess/output/ demoDept2_demoApp2_output.csv				
2	Department History file	C:/eclipse/workspace/WebBasedFormProcess/output/ demoDept2_deptHistory.clp				
3	Department/App History file	C:/eclipse/workspace/WebBasedFormProcess/output/ demoDept2_demoApp2_appHistory.clp				
4	Department	demoDept2				
5	Application Name	demoApp2				
6	Requirements/Suggestions					
7						
8	**Requirement**	**DB Descr**	**Changes DB Desc**	**Form Desc**	**Changes Form**	**Default V**
9	DOC_EXPORTED	DOC_EXPORTED is of type char size 1.				
10	I agree to the below statement:	DOC_AGREE is of type varchar2.		DOC_AGREE will be an Input of type checkbox.		
11	EXPORTED	EXPORTED is of type char size 1.				
12	Were you born in Mississippi?	Were you born in Mississippi? is of type varchar2 size 30.		Were you born in Mississippi? will be an Input of type text.		
13	Last School/College Attended	LASTSCHOOL is of type varchar2 size 128.		LASTSCHOOL will be an Input of type text.		
14	Cancel			The field Cancel will be an Input of type submit.		
15	City of current or last school attended	CURRENTSCHOOLCITY is of type varchar2.		CURRENTSCHOOLCITY will be an Input of type text.		
16	Submit_Application			The field Submit_Application will be an Input of type submit.		
17	City	CITY is of type varchar2.		CITY will be an Input of type text.		
18	Intended audition material:	REPERTOIRE is of type varchar2 size 4000.		REPERTOIRE will be a Textarea.		
19	Email:	CONTACT_EMAIL is of type varchar2 size 255.		CONTACT_EMAIL will be an Input of type text.		
20	SAT/ACT/GRE Score:	SATACTSCORE is of type varchar2 size 10.		SATACTSCORE will be an Input of type text.		
	Have you ever been convicted of a felony or do you currently have felony charges pending	Have you ever been convicted of a felony or do you currently have felony charges pending against you?		Have you ever been convicted of a felony or do you currently have felony charges pending against		

Figure A17. Simplified ".csv" file showing suggested sentences for the new set of requirements when history is utilized.

Once again, novice and expert developer process the suggested requirements shown in Figure A17. The requirements that are irrelevant are eliminated from the list. The requirements that are relevant to the new set of requirements are kept for possible refinement. For instance, the suggested requirement "Email" in row 19 was suggested because the new set of requirements contains "Email address" as a requirement. Each word in "Email address" is matched against each requirement in the knowledge-base. This requirement will be kept and if needed, it will be refined. The refinement process is the same as previously explained when no historical knowledge is present. Based on the expertise of the experienced developer, the novice developer processes and refines all suggested requirements. Once this process is complete, the requirements are then input into the Preliminary Customer Requirement Form. The novice developer meets with the customer for input. If necessary, the entire process may be repeated until requirements satisfy user needs.

REFERENCES

Arikoglu, E. S. (2011). *The Impact of Scenarios and Personas on Requirement Elicitation: an Experimental Study*. Grenoble.

Benitti, F. B. V., & da Silva, R. C. (2013). Evaluation of a Systematic Approach to Requirements Reuse. *J. UCS*, *19*(2), 254–280.

Committee, P. P. (2004). Guide to the Software Engineering Body of Knowledge 2004 Version SWEBOK ® A project of the IEEE Computer Society Professional Practices Committee. *Society*. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4425813

Coulin, C. (2007). A situational approach and intelligent tool for collaborative requirements elicitation, (Toulouse III). Retrieved from http://hal.inria.fr/docs/00/19/58/33/PDF/COULIN_PhD_Thesis_vFinal.pdf

Cybulsky, J., & Reed, K. (2000). Requirements Classification and Reuse: Crossing Domains Boundaries. In *6th International Conference on Software Reuse* (pp. 190–210). Viena: Springer, Lecture Notesin Computer Science.

Davis, A., Dieste, O., Hickey, A., Juristo, N., & Moreno, A. M. (2006). Effectiveness of Requirements Elicitation Techniques: Empirical Results from a Systematic Review. In *14th IEEE International Requirements Engineering Conference (RE'06)2*. IEEE Comput. Soc.

Denger, C., Berry, D. M., & Kamsties, E. (2003). Higher quality requirements specifications through natural language patterns. *Proceedings 2003 Symposium on Security and Privacy*, 80–90. doi:10.1109/SWSTE.2003.1245428

Denger, C., Dörr, J., & Kamsties, E. (2001). A Survey on approaches for writing precise natural language requirements. *Fraunhofer Institut Experimentelles Software*, (070). Retrieved from http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:A+Survey+on+Approaches+for+Writing+Precise+Natural+Language+Requirements#4

Di Stefano, J. S., & Menzies, T. (2002). Machine Learning for Software Engineering: Case Studies in Software Reuse. In *Proceedings of the 14th IEEE international Conference on Tools with Artificial intelligence (ICTAI)* (p. 246). Washington, DC: IEEE Comput. Soc.

Dzung, D. V., & Ohnishi, A. (2009). Ontology-Based Reasoning in Requirements Elicitation. *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, 263–272. doi:10.1109/SEFM.2009.31

Fabrini, F., Fusani, M., Gnesi, G., & Lami, G. (2000). Quality Evolution of Software Requirements Specifications. In *Proceedings of Software and Internet Quality Week*. San Francisco.

Franch, X., Palomares, C., Quer, C., Renault, S., & Tudor, C. R. P. H. (2010). A Metamodel for Software Requirement Patterns, 85–90.

Friedman-Hill, E. (n.d.). Jess Rules. Retrieved from http://jessrules.com/

Friedman-Hill, E. (2003). *Jess in Action: Java Rule-Based Systems*. Manning Publications (July 2003).

Herlea, D., Jonker, C. M., Treur, J., & Wijngaards, N. J. E. (1998). *A Case Study in RE: a Personal Internet Agent*.

Heumesser, N., & Houdek, F. (2003). Towards Systematic Recycling of Systems Requirements. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)* (pp. 512–519). Portland: IEEE Comput. Soc.

HTML Forms and Input. (n.d.). Retrieved from http://w3schools.com/html/html_forms.asp

Humphrey, W. S. (2000). The personal software process (PSP), (November). Retrieved from http://repository.cmu.edu/sei/207/

Humphrey, W. S. (2005). *PSP*. Addison Wesley.

Jacobs, D. (2007). Requirements Engineering So Things Don't Get Ugly. *Engineering*, 8–9.

Jiang, L., Eberlein, A., & Far, B. H. (2004). A Methodology for Requirements Engineering Process Development. In *11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*.

Kaiya, H., & Saeki, M. (2006). Using Domain Ontology as Domain Knowledge for Requirements Elicitation. In *14th IEEE International Requirements Engineering Conference (RE'* (pp. 189–198). Minneapolis/St. Paul: IEEE Comput. Soc.

Kamalrudin, M., Hosking, J., & Grundy, J. (2011). Improving requirements quality using essential use case interaction patterns. *Proceeding of the 33rd International Conference on Software Engineering - ICSE '11*, 531. doi:10.1145/1985793.1985866

Kitazawa, N., Osada, A., Kamijo, K., & Kaiya, H. (2008). So/M: A Requirements De nition Tool using Characteristics of Existing Similar Systems, 255–262. doi:10.1109/COMPSAC.2008.41

Kluge, R., Hering, T., Belter, R., & Franczyk, B. (2008). An Approach for Matching Functional Business Requirements to Standard Application Software Packages via Ontology. *2008 32nd Annual IEEE International Computer Software and Applications Conference*, 1017–1022. doi:10.1109/COMPSAC.2008.147

Knethen, A. V., Paech, B., Kiedaisch, F., & Houdek, F. (2002). Systematic Requirements Recycling Through Abstraction and Traceability. In *Proceeding of Requirements Engineering (RE)*. Essen.

Kohlbacher, F. (2006). The Use of Qualitative Content Analysis in Case Study Research. *Forum Qualitative Social Research*, *7*(1), 1–38. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.1378&amp;rep=rep1&amp;type=pdf

Kroha, P., Janetzko, R., & Labra, J. E. (2009). Ontologies in Checking for Inconsistency of Requirements Specification. In *2009 Third International Conference on Advances in Semantic Processing* (pp. 32–37). Ieee. doi:10.1109/SEMAPRO.2009.11

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys*, *24*(2), 131–183. doi:http://doi.acm.org/10.1145/130844.130856

Leedy, P. D., & Ormrod, J. E. (2009). *Practical Research Planning and Design* (9th ed., pp. 137–138). Old Tappan, NJ: Prentice Hall.

Lopez, M., Moreno, a. M., & Juristo, N. (2000). How to use linguistic instruments for object-oriented analysis. *IEEE Software*, *17*(3), 80–89. doi:10.1109/52.896254

Lorentz, D. (2005). Oracle Database SQL Reference 10g Release 2 (10.2). Retrieved from http://docs.oracle.com/cd/B19306_01/server.102/b14200/title.htm

Mayring, P. (2000). Qualitative Content Analysis. *Forum: Qualitative Social Research [On-Line Journal]*. Retrieved from http://www.qualitative-research.net/fqs-texte/2-00/2-00mayring-e.htm

Mayring, P. (2008). The qualitative content analysis process. *Journal of Advanced Nursing*, *62*(1), 107–15. doi:10.1111/j.1365-2648.2007.04569.x

Mili, H., Mili, F., & Mili, A. (1995). Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, *21*(6), 528–562.

Noy, N. F., Fergerson, R. W., Musen, M. A., & Informatics, S. M. (2000). The knowledge model of Protégé-2000   : combining interoperability and flexibility. *Knowledge Engineering and Knowledge Management Methods Models and Tools*, *1937*(1), 1–20. Retrieved from http://www.springerlink.com/index/ff4979945txfvkku.pdf

Noy, N. F., & McGuiness, D. L. (2001). *Ontology Development 101: A guide to creating your first ontology*. Stanford, CA.

Ohnishi, A. (1994). Customizable Software Requiremenst Languages. In *Proceedings of the 8th International Computer Software and Application Conference (COMPSAC)*. Los Alamitos: IEEE.

Omoronyia, I., Sindre, G., & Stålhane, T. (2010). A Domain Ontology Building Process for Guiding Requirements Elicitation. *Springer*, 188–202. Retrieved from http://www.idi.ntnu.no/grupper/su/publ/stalhane/inah-refsq10.pdf

Oracle. (n.d.). Oracle SQL Developer Documentation. Retrieved from http://docs.oracle.com/cd/E35137_01/index.htm

Pfleeger, S. L., & Atlee, J. M. (2006). *Software Engineering Theory and Practice* (3rd ed., pp. 141–222). Pearson Education, Inc.

Rolland, C., & Proix, C. (1992). A Natural Language Approach for Requirements Engineering. *Pp. 257-277 in Proceedings of Conference on Advanced Information Systems Engineering, CAiSE 1992, Manchester, UK 12-15 May*.

Souag, A. (2012). Towards a New Generation of Security Requirements Definition Methodology Using Ontologies. In *24th International Conference on Advanced Information Systems Engineering (CAiSE'12)*. Gdansk, Poland.

Toval, A., Nicolás, J., Moros, B., & García, F. (2002). Requirements Reuse for Improving Information Systems Security: A Practitioner's Approach. *Requirements Engineering*, *6*(4), 205–219. doi:10.1007/PL00010360

Williams, L. A. (2000). *The Collaborative Software Process*. The University of Utah.

Wilson, W. M. (n.d.). Automated Quality Analysis of Natural Language Requirement Specifications.

Zhang, Z. (2007). Effective Requirements Development - A Comparison of Requirements Elicitation techniques. In *INSPIRE* (pp. 225–240). Tempere, Finland.

Zong-yong, L., Zhi-xue, W., Ying-ying, Y., Yue, W., & Ying, L. (2007). Towards a Multiple Ontology Framework for Requirements Elicitation and Reuse. In *In COMPSAC* (pp. 189–195).