UNIVERSITY OF CALIFORNIA

SANTA CRUZ

A Representation-Based Methodology for Developing High-Value Knowledge Engineering Systems: Theory and Applications

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

TECHNOLOGY AND INFORMATION MANAGEMENT

by

Tyler Munger

June 2012

The Thesis of Tyler Munger
is approved:

_____

Subhas Desa, Chair

_____

Patrick Mantey

_____

Arnav Jhala

_____

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

A Representation-Based Methodology for Developing High-Value Knowledge Engineering Systems: Theory and Applications

by

Tyler Munger

Nearly all enterprises are routinely collecting data and information and attempting to transform it into knowledge that can be applied to core business activities, e.g. product development, customer support, and marketing. In this work we develop and apply a representation-based methodology for building Knowledge Engineering systems to support the rapid and effective extraction of knowledge from unstructured data. The proposed methodology draws upon methods and techniques from Knowledge Engineering, Product Design, and Software Engineering in order to maximize the overall value of the system with respect to the needs of the users and organization. The domain of Product Design provides formal tools for identifying user needs, exploring different function realizations, and managing trade-offs between quality and cost. Application of the methodology at a large computer networking company produced a "Service Request Portal" tool that was well received by end-users and resulted in a 30% productivity improvement compared to previous tools.

DEDICATION

This thesis is dedicated to my parents, Don and Rebecca Munger. Without their love and support, this thesis would never have been possible.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Subhas Desa for giving me the opportunity to work on this thesis. I have had the privilege of working with Subhas as both an undergraduate and graduate student, and I could not have asked for a better mentor. I am very grateful for the knowledge and wisdom he has shared with me and I am looking forward to many more years of collaboration. Subhas is also responsible for one of the core ideas in this thesis, the Integrated Meta-Representational Model, which provided the theoretical framework for formalizing the integration of Knowledge Engineering, Product Design, and Software Engineering.

I would like to thank the Smart Call Home group at Cisco Systems for providing me the opportunity to develop and test the ideas in this thesis within the context of a real-world Knowledge Engineering problem. In particular, I owe special thanks to Sri Ramachandran for his support and collaboration. I would also like to thank Danny Core who helped develop several key components of the Knowledge Engineering system described in this thesis including the Graphical User Interface.

None of this work would have been possible without the support of the Network Management Operations Lab. In particular, I would like to thank Patrick Mantey and Brad Smith who have been invaluable sources of knowledge, encouragement, and support over the last four years. I would also like to thank Chris Wong at Cisco for his patience and understanding while I finished this thesis.

I would also like to thank Patrick Mantey and Arnav Jhala for serving as members of my thesis committee. I greatly appreciate the time they have spent reading drafts and providing feedback.

Finally, I would like to thank to my friends and family for bearing with me through the last couple of months; I know that I haven't been easy to be around. Karla, I am especially grateful for your patience, love, and support.

1 Introduction

1.1 Background

Nearly all enterprises, technology and otherwise, are routinely collecting data and information and attempting to transform it into knowledge that can be used to influence core business activities, e.g. product development, customer support, and marketing. Despite advances in text and data mining, the process of extracting useful knowledge from complex unstructured data almost always requires human domain experts (knowledge workers). The development of software-based Knowledge Engineering (KE) systems to support these knowledge workers with the rapid and effective extraction of knowledge from this massive ongoing collection of data would be a key source of competitive advantage, in particular enabling these enterprises to develop smarter, more customer-centric, products and services.

This thesis addresses how enterprises can develop high-value software-based KE systems for supporting the knowledge extraction process. It is important to differentiate the "modern" KE systems addressed in this thesis from "classical" KE systems such as expert systems. Classical KE systems primarily address the problem of codifying human knowledge so that can be reused throughout the organization. We are interested in "modern" KE system that provide software automation to support existing work process and enable knowledge workers to efficiently work with massive amounts of data. Because modern KE systems involve integrating software systems into existing work processes, they share a number of similarities with Decision Support Systems (DSSs). However, DSSs generally operate with structured data, while modern KE systems support processes involving unstructured data. Table 1.1 highlights some of the key differences between classical KE systems, modern KE systems, and DSSs.

Table 1.1: Comparison of modern Knowledge Engineering systems, classical Knowledge Engineering systems, and Decision Support Systems

	Modern Knowledge Engineering System	Classical Knowledge Engineering system	Decision Support System
Knowledge source	Unstructured data	Domain experts	Structured data
Type of problems	Unstructured	Structured	Semi-structured
Users	Domain experts	Non-experts	Managers
Level of interactivity	High	Low	High
Computational requirements	Medium - high	Low - medium	Medium - high
Development challenges	Work-process integration (usability, adoption, etc.)	Knowledge capture and representation	Data integration

1.2 Research Issues

The development of modern Knowledge Engineering (KE) systems can be separated into two distinct but related research areas:

1. **Software Infrastructure for Knowledge Engineering**: developing KE systems that are high-quality software products which generate value to the end-users and organization with respect to impact and cost.

2. **Analytics for Knowledge Engineering**: layering analytics, based on Data Mining and Information Retrieval tools and techniques, on top of the software infrastructure in order to extract useful knowledge from large quantities of unstructured data.

The emphasis of this thesis is on research area (1): the development of software infrastructure for Knowledge Engineering systems.

Modern KE systems are interactive software products for helping knowledge workers efficiently extract knowledge from data and information. Three important requirements for successful Knowledge Engineering (KE) system development are as follows: First, the KE system must be tightly integrated into the existing work processes in order

to maximize the overall productivity of the end-users. Second, the KE system must be a high-quality product—reliable, easy to use, attractive—in order to be adopted by the end-users (knowledge workers). Third, the KE system must be high-impact and low-cost in order to a good investment for the organizations. Addressing of these three requirements requires a multi-disciplinary approach based on the following three domains:

- **Knowledge Engineering**: provides tools for modelling end-users work processes in order to tightly integrate the system into existing work processes.
- **Product Design**: provides formal tools for explicitly identifying end-user needs for the system, exploring different function realizations, and managing the inevitable trade-off between quality and cost.
- **Software Engineering**: provides tools and techniques for efficiently developing robust and reliable software systems.

While there are a number good methodology for classical KE system development, such as CommonKADS [Schreiber, 1994] and MIKE [Angele et al., 1992], there is a lack of mature methodologies for developing modern Knowledge Engineering systems. Consequently, the selection and application of methods from these three domains is often ad-hoc, and, being ad-hoc, suffers from a number of issues including: insufficient capture the end-users' existing work-process, focus on the technical aspects of the system rather than users' needs, and lack of user involvement during system development. The combination of these factors, in particular the lack of attention to the user and organizational needs for the system, frequently results in the deployment of these system not yielding useful results.

The objective of this thesis is to create an end-to-end process methodology, based on integrating the appropriate KE, PD, and SE tools, for developing high-value modern

KE systems. To this end, the following research issues will be addressed:

1. What is a rational model for organizing the different activities—defining requirements, system design, software development, testing, etc.—involved in KE system development?

2. What methods and techniques from KE, PD, and SE are necessary when developing KE systems in a high-technology context, e.g. computer networking products and technologies?

3. How should the necessary methods and techniques be integrated into an process methodology for KE system development?

4. What are the benefits of an integrated process methodology when developing KE systems? What are the critical set(s) of methods for scenarios when it is either beneficial or necessary to simplify the application of these methods?

1.3 Research Contributions

This thesis makes two key research contributions to the theory and practice of Knowledge Engineering (KE) system development:

1. An **Integrated Meta-Representation Model (IMRM)** for structuring the KE system development process into five levels of representation. (Theory)

2. An **Integrated Representation-Based Process Methodology (IRPM)**, based on the IMRM, that integrates methods and techniques from the domains of Knowledge Engineering, Product Design, and Software Engineering into a unified framework for KE system development. (Practice)

The IMRM is based on the following fundamental notions in cognitive neuroscience: the human brain is representational system; the human-brain solves problems by devel-

oping representations appropriate for solving the problem ([Metzinger, 2003], [Revonsuo, 2009]). By formalizing these notions, the IMRM provides an integrated representational framework that enables the use and integration of representational tools from different domains intro a comprehensive process methodology (IRPM) for end-to-end development of Knowledge Engineering systems.

1.3.1 Integrated Meta-Representational Model

The Integrated Meta-Representational Model consists of five interconnected levels of representation (shown in Figure 1.1). The first level of representation, the *External*, models the current organizational environment where the system will deployed and defines the initial state for the development process. The next level of representation, the *Outside-In* determines the user needs (requirements) for a Knowledge Engineering (KE) system and defines the goal state for the development process. The Internal level of representation takes us inside the system and addresses the conceptual design issues related to how the system will satisfy the user needs subject to the organizational constraints (cost, development time, etc.). The *Inside-Out* level of representation takes us outside the Internal and addresses the translation of the conceptual design into a software design. Finally, the *Outside* level of representation is the software implementation of the KE system that realizes the goal state. Together, the five levels of representation take us through the entire KE system development process; from user requirements to robust end-product.

The integrated representational structure of the IMRM allowed us to answer research questions (1) and (2):

1. **What methods and techniques are necessary when developing Knowledge Engineering systems?** IMRM provides the selection criteria that allows us to identify the necessary representational from the domains of Knowledge Engi-

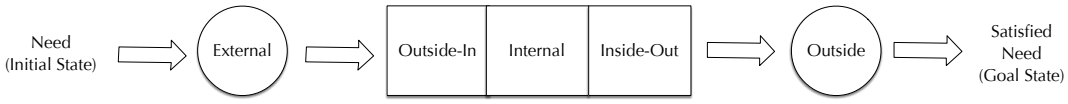

Figure 1.1: Integrated Meta-Representational Model

neering, Product Design, and Software Engineering. A description of the methods and techniques that were selected from each domain is provided in Section 4.

2. **How should the necessary methods and techniques be integrated?** Multidisciplinary development is notably difficult because different domains tend to conceptualize and represent knowledge differently. The IMRM provides the common representational language necessary for integrating methods across multiple domains into unified development process. A description of this integration is discussed in Section 5.

The development of software-based Knowledge Engineering systems naturally involves work in the Knowledge Engineering (KE) and Software Engineering (KE) domains. However, existing approaches to KE system development, such as the CommonKADS methodology [Schreiber, 1994], do not sufficiently address the interface between these two domains. The IMRM provides a consistent representational framework for the entire development process and, thereby, enables a more seamless transition between the KE and SE domains. Smoothing the transition between these domains minimizes rework, reduces development time, and generally increases the probability that the end-product will match the original requirements.

The IMRM also exposed an important gap in the KE and SE tools for representing

the system at the Internal level of representation. Without the appropriate tools for Internal representation, the creative process of translating user needs into functional specifications, exploring the space of possible solution-principles, and selecting the best mix of solution-principles is largely dependent on the individual effort, skill, and experience of the designer. The integration of conceptual design tools from the domain of Product Design, such as Function Structures, Morphological Matrices, and Utility Functions, formalizes this process and moves it away from intuitive ("ad-hoc") methods to robust, repeatable methods that allow a wide range of designs to be quickly generated and evaluated. Systematic generation of the design will, in general, produce a higher-quality design with respect to the user and organizational needs.

1.3.2 Integrated Representation-Based Process Methodology

The Integrated Representation-Based Process Methodology (IRPM) brings together representational tools from Knowledge Engineering (KE), Software Engineering (SE), and Product Design (PD) into an end-to-end framework for developing Knowledge Engineering systems. The KE domain provides the modelling tools for understanding the organizational context of system and work process of the end-users. The PD domain provides formal tools for explicitly identifying user needs for the system, exploring different function realizations, and managing the inevitable trade-offs between a high-quality system that satisfies the end-user needs and minimizing development costs (time and money). Lastly, the SE domain provides the tools and techniques necessary for rapidly developing robust and reliable software systems. Figure 1.2 shows a high-level view of the how the different KE, SE, and PD methods are integrated in the IRPM.

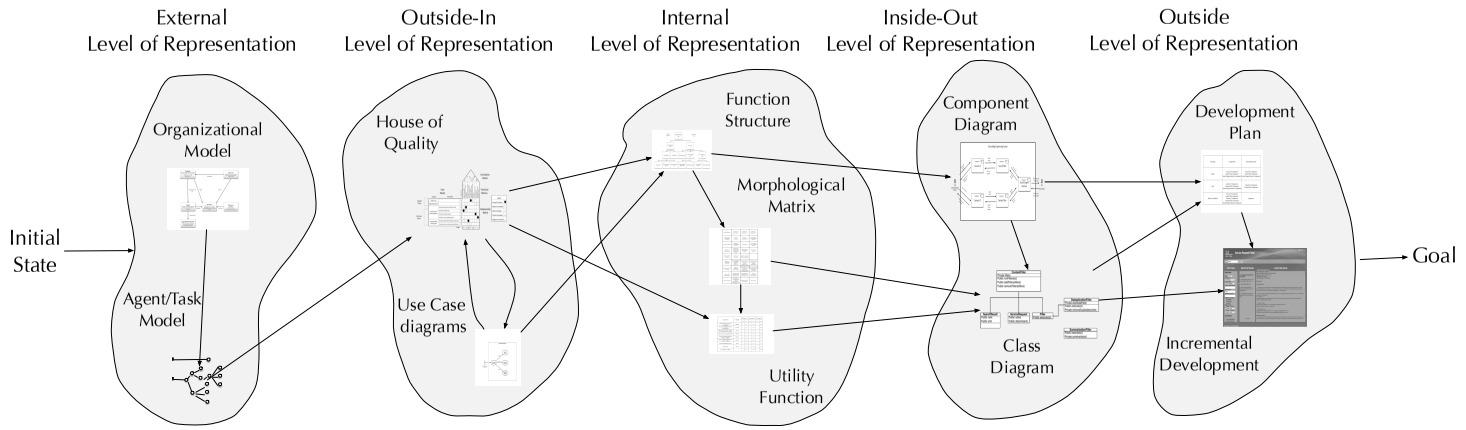

Figure 1.2: Integrated Representational Process Methodology

We have used the IRPM to develop a simple but non-trivial KE system for working with service request (support case) data in the computer networking domain. The resulting system improved productivity by over 30% compared to the previously used tools, was well received by the users, and was developed within the organizations budget and schedule constraints. Reflecting on the development process allowed us to answer research question (3): What are the benefits of an integrated process methodology when developing Knowledge Engineering systems?

- **Requirements Analysis**: The IRPM allows for a more efficient Requirements Analysis work flow by combining UML Use Cases [Schach, 2008] with a Product Design technique known as the House of Quality [Hauser and Clausing, 1988]. The House of Quality provides a prioritized set of user needs for guiding the process of creating Use Case and ensured that each Use Case was directly related to an important user needs. By focusing Use Cases on user needs the IRPM minimizes the number of iterations necessary to produce a minimal, complete, and unambiguous set of requirements that accurately reflect the user needs for the system. (See section 5.3).

- **System Design**: The IRPM allows for value (ratio of function-to-cost) to be explicitly addressed during the system design work flow by supplementing standard Software Engineering methods with a conceptual design "front-end". This conceptual design "front-end" uses Product Design techniques— such as Function Structures [Pahl and Beitz, 1996], Morphological Matrices [Pahl and Beitz, 1996], and Utility Functions [Cross, 1998]—to efficiently explore the design space defined by the user needs. Systematic exploration enables a wide range of design approaches to be generated and evaluated before transitioning to the Software Engineering methods. As a result, the IRPM maximizes the probability of finding the best design, with respect to the user and organizational needs for the system. (See section 5.5).

- **System Development**: The IRPM allows for more effective user participation during system development work flow because the House of Quality enables clear targets to be set for what is expected of the system during the system development. These targets provide a framework for guiding how users tested the system, interpreting the feedback, and using it to drive the prototyping process. (See section 5.6).

Comparison of the developed system with an Analytics for Knowledge Engineering based approach [Wang et al., 2010] to a similar Knowledge Engineering problem in the computer networking domain, shows that a Software Infrastructure for Analytics approach allowed us to achieve comparable results using significantly less complex analytical (Data Mining/Information Retrieval) components.

1.4 Organization of the Work

The thesis is organized as follows: Chapter 2 establishes the need for a multidisciplinary Knowledge Engineering (KE) system development methodology and outlines the problems involved in developing such as methodology. Chapter 3 surveys the work we are drawing upon in order to create our KE system development methodology. Chapter 4 explains the Integrated Meta-Representational Model (IMRM) for structuring the KE system development process and describes selection of the necessary representational tools for the Integrated Representation-Based Process Methodology (IRPM) for KE system development. Chapter 5 walks through the five levels of the IRPM. Chapter 6 presents the results of applying the IRPM to a real Knowledge Engineering problem at a large computer networking company. Chapter 7 discusses the impact of the different domains on the KE system development process, compares the IRPM with an "Analytics for Knowledge Engineering" approach. Our conclusions and several possible paths for future work and then presented in Chapter 8.

2 Problem Description

In this chapter we describe Knowledge Engineering (KE) system development process and establish the need for a multidisciplinary approach. To this end we start by explaining the organizational need that leads to the development of a modern KE system. We then use a real example in the computer networking domain to concretely illustrate a typical initial state, goal state, and the issues involved in going from this initial state to the goal state. An analysis of these issues motivates the need for a multi-disciplinary approach to KE system development. Lastly, we outline the tasks involved in developing a process methodology to support a multi-disciplinary approach to KE system development.

2.1 Knowledge Engineering System Development: Motivation

Nearly all enterprises, technology and otherwise, are routinely collecting data and information as part of the ongoing process of conducting business. In general the collected data can be organized into two high-level categories: transactional data and interactional data [Spangler and Kreulen, 2008]. Transactional data is produced by transactions with customers, e.g. point-of-sale, and is generally structured. An example of transactional data would be a list of products that were sold over the last month. Interactional data is produced by interactions with customers, e.g. customer support cases, and is generally unstructured (free-form text). An example of interactional data would be the support cases received by a service center over the last month. The extraction of knowledge from these both transactional and interactional data would be a key source of competitive advantage, in particular enabling these enterprises to develop smarter, more customer-centric, products and services.

The problem of mining knowledge from transactional data is well understood [Witten and Frank, 2005]. Data Mining algorithms generally perform well on transactional data

and organizations have been developing been applying these algorithms in practice for some time. One well-known example knowledge extraction from transactional data is market basket analysis where co-occurrences in historical customer purchases are used in order to recommend new products to customers.

The extraction of knowledge from interactional data, however, is a far more difficult problem and usually requires human domain expertise. In most organizations this process is still largely manual (Figure 2.1). Knowledge workers read through interactional data in order to identify useful knowledge, structure it, and then add it to a database where it can be used to support core business activities such as product development, customer support, and marketing.

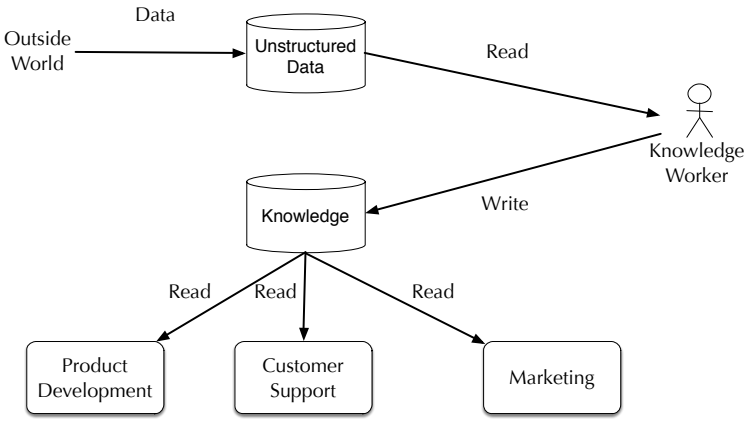

Figure 2.1: Manual knowledge extraction process

Manual knowledge extraction is tedious and inefficient. Knowledge workers spend a significant amount of their time and effort reading through irrelevant and poorly formatted data. Furthermore, as the amount of information being collected by enterprises grows, it becomes increasingly difficult for knowledge workers to process all of the data.

In order to enable efficient and effective extraction of knowledge from interactional data it is necessary to support the extraction process with software-based Knowledge Engineering (KE) systems.

2.2 Knowledge Engineering System Development: An Example

Network service centers receive thousands of customer support cases every day on a wide variety of product problems. Each of these support case is tracked by a service request document that contains the complete transcript—emails, phone conversations, etc.—of all the customer's interaction with the service center's Technical Support Engineers (TSEs). By the time a typical support case is closed, the associated service request document contains 30-50 pages of free-form text. Buried within this massive collection of free-form text is useful knowledge about product problems encountered by customers and solutions to these problems created by TSEs.

Network Knowledge Engineers (NKEs) mine resolved service requests for problem-solution pairs that can be applied to areas such as new product design and product support. Figure 2.2a shows the high-level flow of the NKE work process for creating problem-solution pairs. NKEs start with a particular product problem for which the organization needs solutions. For example, a router crashes when a particular routing protocol, e.g. OSPF, is enabled. The NKEs then use keyword queries, e.g. OSPF crash, to search for service requests that are potentially relevant to the product problem of interest. Each set of service requests returned by the search engine is first briefly read to establish relevance. Once a service request is determined relevant, then the NKEs will read through the service request in detail to extract the solution prescribed by the TSE. When a sufficient number of solutions have been collected, the NKEs formulate a generalized problem-solution pair that contains the steps for resolving the product problem.

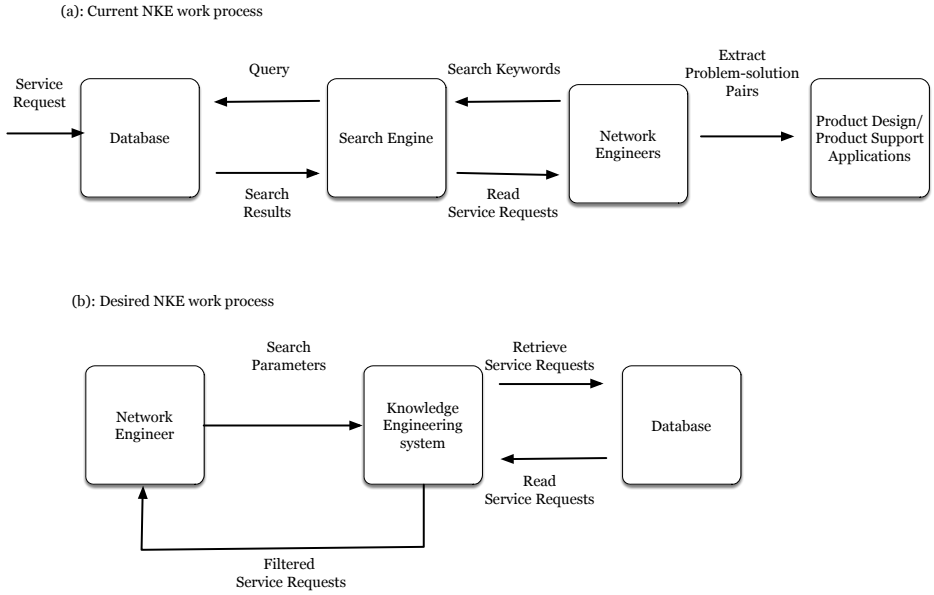

Figure 2.2: Network knowledge engineer work process

The process of extracting problem-solution pairs from service requests is complicated by the following two issues. First, the relevance of a service request depends on a number of attributes (attachments, tags, etc.) that are not captured during keyword search. This causes the keyword queries that the NKE use when searching for service requests to return a large number of irrelevant results that must be manually evaluated for relevance. Second, each service request contains 30-50 pages of unstructured technical documentation describing the customer problem and recommended solution. As a result, the NKEs spend a significant amount of time and effort looking for information that is hidden within irrelevant email threads, poorly formatted text, and duplicated content. The combination of these two issues is a serious bottleneck in the NKEs work process. NKEs spend the majority of their time searching for service requests and extracting solutions, rather than formulating problem-solution pairs. At the organizational level these problems make the development of problem-solution pairs both costly

and inefficient.

In order to make the process of generating problem-solution pairs scalable and cost effective, the organization would like to develop a KE system that improves the productivity of the NKEs. Figure 2.2b shows a high level view of the desired goal state where the work process would be supported by a KE system. NKEs provide the system with a set of search parameters that precisely describe the types of service requests that they were interested in locating. The system then retrieves a set of service requests that meet then these criteria. When the NKEs select a service request for reading, the system retrieves it from the database and automatically restructures the text so that it is easier to find the relevant information.

In order to develop the desired KE system we first need to capture the users (NKEs) work process and use it to determine what area(s) of the work-process the system should support. Next, we need to work with the NKEs to define a detailed functional specification of what the system will need to do, e.g. how should the system filter the service request content. We then need to determine the best approach for implementing the functional specifications based on the user needs and the organizations objective for the system. This approach then must be translated into a software design. Additional issues that need to be addressed when creating the software design include: designing a user interface for the system, making the system robust and reliable, and modularizing the system so that it maintainable. Next we need to develop the KE system based on the software design and ensure that it is free of bugs and software defects. Finally once the system is developed we need to test it with users and evaluate the impact of the system.

2.3 Knowledge Engineering System Development: Issues

The desired overall goal of successful Knowledge Engineering system development is to create a system that is valuable to both the users and the organization. In order to achieve this goal the following issues must be addressed:

- Tight engagement with the end-users by modelling and representing user work process and requirements.
- Making the system useful and attractive to users.
- Balancing user needs (e.g. high-quality) with organizational needs (e.g. low-cost).
- Involving the user in prototyping. Testing and refining the system with user participation.
- Ensuring the system is robust and reliable.

2.4 The Need for an Integrated Multidisciplinary Approach

It quickly becomes apparent that addressing the described issues requires a multidisciplinary approach involving work in the following domains: Knowledge Engineering, Product Design, and Software Engineering. First and foremost, tools and techniques from the domain of Software Engineering are needed to efficiently develop robust and reliable KE system software implementations. However, the domain of Software Engineering does not contain tools for understanding the wider organizational context for the system and ensuring that the system is sufficiently integrated into existing work processes. Therefore, tools from the Knowledge Engineering domain are necessary for modelling the organizational context of system and work process of the end-users. Finally, KE systems are interactive products that will be used by knowledge workers as part of their daily work process. This requires tools and techniques from the domain of

Product Design to explicitly identify user needs for the system, exploring different function realizations, and manage the inevitable trade-offs between a high-quality system that satisfies the end-user needs and minimizing development costs (time and money).

Without a structured process for handling the integration of Knowledge Engineering, Product Design, and Software Engineering the selection and application of methods from these three domains is often ad-hoc, and, being ad-hoc, suffers from a number of problems including: insufficient capture the end-users' existing work-process, focus on the technical aspects of the system instead of users' needs, and lack of user involvement during system development. The combination of these factors, in particular the lack of attention to the user and organizational needs for the system, frequently results in the deployment of these system that fail to yield useful results.

This thesis addresses the problem of integrating the appropriate Knowledge Engineering, Product Design, and Software Engineering tools into an end-to-end process methodology based on for developing high-value Knowledge Engineering (KE) systems. To this end, the following tasks will need to be addressed:

1. Develop a rational model for structuring the different activities involved in KE system development. (See Chapter 4)

2. Use the developed model to select the most useful set of methods and techniques from the KE, PD, and SE domains. Chapter 4)

3. Integrate the selected methods into a process methodology for KE system development. (See Chapter 5)

4. Determine the benefits of an integrated process methodology and identify critical paths for scenarios when it is either beneficial or necessary to simplify the process methodology. (See Chapters 6, 7)

3 Related Work

In this chapter we review related work to our objective of developing high-value Knowledge Engineering (KE) systems. We start with a taxonomy of related work in the areas of Knowledge Engineering, Product Design, and Software Engineering. We then provide a brief an overview of the related work that we are drawing upon in order to create our process methodology. Lastly we describe other research which addresses similar issues to those described in the Problem Description (Chapter 2).

3.1 Taxonomy of Related Work

We have organized related work according to two high-level categories: works related to creating an end-to-end process methodology for Knowledge Engineering system development, and works related to the problems and issues involved in Knowledge Engineering system development. Figure 3.1 shows methods and techniques in each category.

3.2 Work Related to Developing the Thesis Methodology

In order to create an end-to-end process methodology for Knowledge Engineering system development we draw upon work in the domains of Knowledge Engineering (KE), Product Design (PD), Software Engineering (SE). The KE domain provides modelling tools for understanding the organizational context of system and work process of the end-users. The PD domain provides formal tools for explicitly identifying user needs for the system, exploring different function realizations, and managing the inevitable trade-offs between a high-quality system that satisfies the end-user needs and minimizing development costs (time and money). Lastly, the SE domain provides tools and techniques necessary for rapidly developing robust and reliable software systems. The following sections provide a high-level overview of the methods and techniques in each

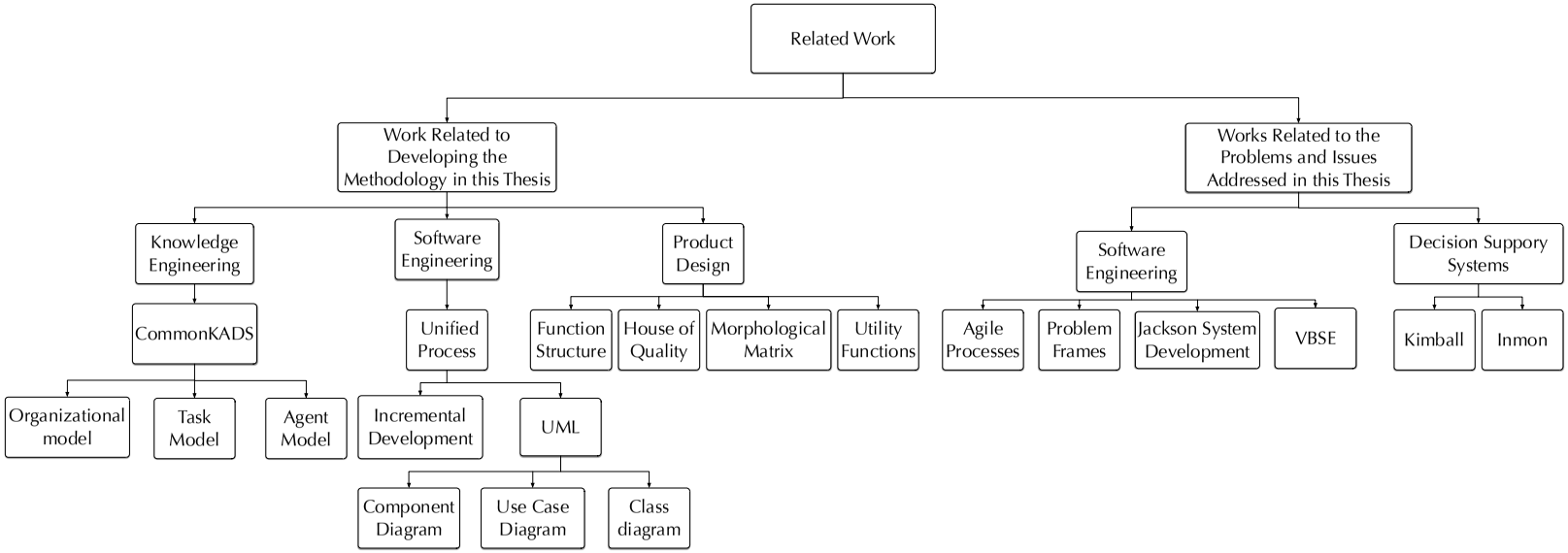

Figure 3.1: Taxonomy of related work in Knowledge Engineering, Product Design, Software Engineering

domain that form the core of our process methodology.

3.2.1 Knowledge Engineering

Within the Knowledge Engineering domain we have drawn tools from the CommonKADS methodology [Schreiber, 1994]. The CommonKADS methodology originated out of the need to build classical Knowledge Engineering (KE) systems, e.g. expert systems, on a large scale in a structured and repeatable way. In order to support this objective CommonKADS provides three perspectives (sets of models): Context, Concept, and Artifact. The Context perspective addresses the organizational environment in which the system will operate and is used to understand the objectives for the KE system and how it will fit into existing processes. The Concept perspective addresses the knowledge component of the system and is used to identify and capture the knowledge necessary to solve a particular task. The Artifact perspective addresses the design of the system and is used to specify the system architecture and computational mechanisms. Figure 3.2 shows the models at each perspective and their relationships.

Although the CommonKADS methodology was originally developed for supporting the development of classical (expert system) KE systems, it contains a number of useful tools that can be re-purposed to support the development of modern KE systems. Both classical and modern KE systems must function within the context of the overall organization. To this end, we are drawing upon the three models in the CommonKADS Context perspective in order to address the organizational factors involved in KE system development. The Organizational model is used to understand the organizational context for the KE system and ensure that the system is aligned with the organizational needs (See Section 5.2.1). The Agent and Task models is used to zoom in and document the work-processes in this context (See Section 5.2.2). (Since it is not necessary to build a formal model of the expert knowledge when developing modern KE systems, we are not be using any of the CommonKADS models from the Concept and Artifact

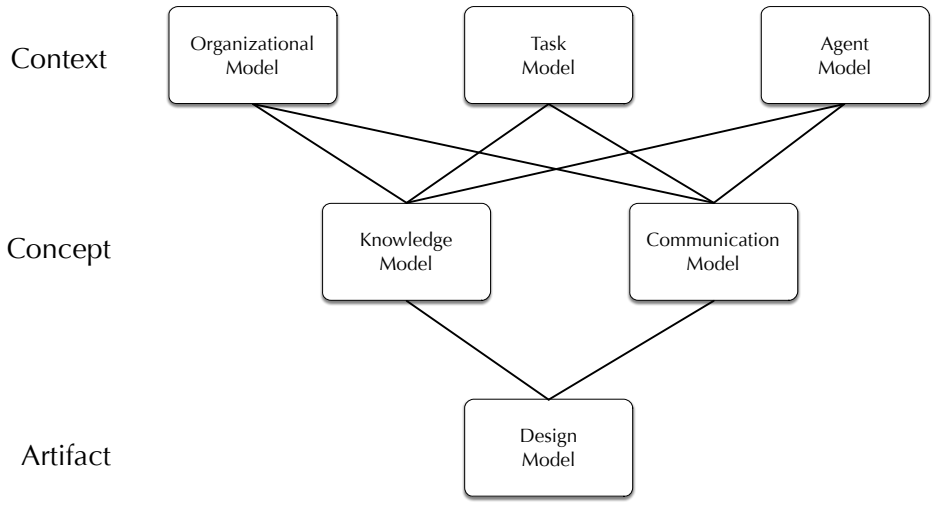

Figure 3.2: Perspectives and models in the CommonKADS methodology [Schreiber, 1994]

perspectives.)

3.2.2 Product Design

In the domain of Product Design we have drawn upon four well-known techniques for conceptual design. The House of Quality [Hauser and Clausing, 1988] is used to translate the user (customer) needs into measurable engineering characteristics that can be used by the development team to design the system. The Function Structure [Pahl and Beitz, 1996] is used to specify the functions and sub-functions that the system must perform (See Section 5.4.1). The Morphological Matrix [Pahl and Beitz, 1996] technique is used to explore the space of possible solution-principles (realizations) to a set of functional specifications (See Section 5.4.2). Finally, the Utility Function [Cross, 1998] technique provides an objective way of selecting the design that best satisfies the customer needs

and other objectives (costs, time, etc.) for the product (See Section 5.4.3).

3.2.3 Software Engineering

Within the Software Engineering domain we have drawn upon methods from the Unified Process [Schach, 2008], a well-known and widely used Software Engineering methodology for end-to-end development of large software systems. The Unified Process (UP) is an iterative, architecture-centric, and use-case driven methodology. Iterative means that system functionality is delivered in chunks, or increments, leading to a fully functional system. Architecture-centric means that system architecture is defined early in the development process and then used to guide the other development activities. Lastly, Use Case driven means that the all system functionality is derived from Use Cases.

The UP is divided into four high-level phases: Inception, Elaboration, Construction, and Transition. The Inception phase identifies the initial set of requirements, the business case, and outlines the scope of the system. The Elaboration phase the expands upon the results of the Inception phase in order to capture the a full set of requirements for the system. The Construction phase addresses the development of the system as a series of short, time-boxed iterations. Lastly, the Transition phase is where the system is deployed to the end-users. Figure 3.3 shows the four phases in the UP and the work flows involved in each phase.

The UP is comprehensive end-to-end software development methodology. We have drawn on a subset of the tools from the Elaboration and Construction phases that are most useful KE system development. In the order to document the system requirements we have drawn upon the UML Use Case diagram 5.3.2 to capture how the end-users will interact with the system. In the area of software design we have adopted the UML Component, and Class diagrams to guide the software design process (See Sections 5.5.1 and 5.5.1). In the area of software development we have drawn upon the Incremental

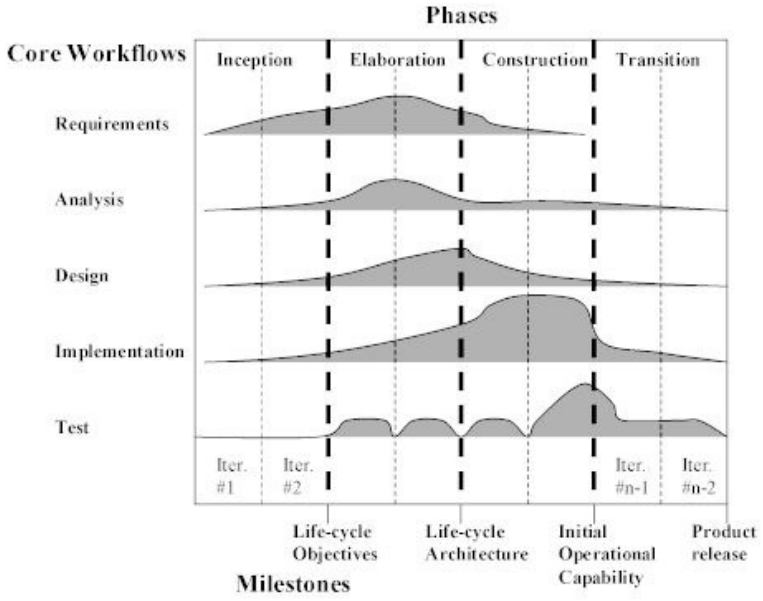

Figure 3.3: Four phases of the Unified Process [Schach, 2008]

Development approach where the system is incrementally developed through a set of build and test cycles (See Section 5.6.2).

3.3 Work Related to the Problems and Issues Addressed in this Thesis

A number of the issues brought up in the Problem Description (Section 2) are not exclusive to Knowledge Engineering systems. In this section we briefly describe other research in the areas of Software Engineering and Decision Support Systems that addresses similar research issues.

3.3.1 Software Engineering

In this section we briefly describe other related research in the domain of Software Engineering and evaluate their applicability to modern KE system development.

Agile Process

Successful Knowledge Engineering (KE) system development requires a high-level of end-user participation in order to achieve a tight integration with existing work processes. Agile Processes [Schach, 2008] are a group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration with the customers (end-users). Notable methodologies that fall under the umbrella of Agile Processes include: Extreme Programming (XP), Crystal, and Scrum. In the Extreme Programming methodology customers are presented with a set of features and cost (time and money) estimates. The customer then selects which features to include in each development iteration. Crystal focuses on developing team structures that minimize the need for formal processes during development. Scrum breaks the development process into 30 day sprints and uses self-organizing teams to keep track of progress.

Although Agile Processes work well for small software projects, they have three major disadvantages when applied to large complex systems. First, Agile Processes require that customers (end-users) have a clear vision of the end product and can easily loose its direction if customers have only vague idea of the desired product. Second, Agile Processes build the system incrementally and do not provide adequate support for developing an overall architecture for the system. Finally, Agile Processes rely on informal communication between team members and do not scale well to larger teams.

Jackson System Development/Problem Frames

Tight integration with existing work processes also requires that the development process be driven by a model of the real-world. The Jackson System Development (JSD) [Jackson, 1983] approach was one of the first Software Engineering methodologies to recognize the importance of modelling real world (the problem space) before starting software design and development (the solution space). To this end, JSD provides the entity structure diagram tool for describing the aspects of the business or organization that the system will be concerned with. Jackson later expanded on the ideas in JSD with the Problem Frames [Jackson, 2001] approach that provides a more complete set of tools for describing software development problems. Each Problem Frame describes a problem as consisting of the software machine and its relationship to one or more application domains. Each class of problems is called a problem frame (roughly analogous to a design pattern).

Although JSD and Problem Frames both contain useful tools for modelling software problems, neither of them are very practical for KE system development. JSD was developed in the 1980s and does not provide support for modern Software Engineering paradigms such as Object Oriented development. On the opposite end of the spectrum, Problem Frames is still a relatively new approach and hasn't been integrated into standard practices, e.g. UML, which makes it difficult to use with conventional Software Engineering methodologies such as the Unified process.

Value Based Software Engineering (VBSE)

A key challenge when developing KE systems is creating a system that is valuable to both the end-users and the organization. Conventional Software Engineering methodologies, such as the Unified Process, operate in a value-neutral setting where every use

case and requirement is equally important. Value Based Software Engineering (VBSE) [Boehm, 2003a] brings value considerations to existing Software Engineering methodologies. VBSE is based around seven key practices such as Business Case Analysis, Concurrent Engineering, and Agile development. Together these practices provide a framework for managers and software designers/developers to make decisions that generate better value for the customers (end-users) and reduce wasted effort.

VBSE introduces many useful practices for how value consideration should be integrated into software development, however, it lacks the tools for implementing these practices within the context of conventional Software Engineering methodologies, e.g. the Unified Process. VBSE practices, such as Concurrent Engineering, are described as high level objectives of what should be done but not necessarily how to do it. While these high-level descriptions might be sufficient when developing small systems with relatively simple value considerations, they are do not provide adequate support for KE system development.

3.3.2 Decision Support Systems

Decision Support Systems (DSSs) support organizational decision-making and problem solving by allowing users to rapidly analyze large quantities of data. DSSs are similar to modern KE systems in that they involve the integration of a software system into existing work processes. Therefore, DSS development must consider issues many of the same issues as modern KE system development such as user involvement during development and system usability. However, DSSs focus on structured (transactional) data while KE systems focus on unstructured (interactional) data. As a result, DSSs are often used by at the managerial level for evaluating different scenarios, whereas modern KE systems have a broader range of applications. We will focus on recent work in the area of Data-Driven Decision Support Systems (DDSSs). DDSSs address the integration transactional data from multiple sources into data warehouse which can be used to

support applications such as project planning, supply chain management, marketing. There are two dominant approaches for building DDSSs: the Inmon approach [Inmon, 2002], and the Kimball approach [Kimball and Ross, 2000].

Inmon Approach

The Inmon approach [Inmon, 2002] takes a top-down approach to developing DDSSs involving three levels of data modelling. At the first level of modelling, Entity Relationship Diagrams (ERDs) are used to define the data that the organization is collecting. The ERDs are then consolidated into a centralized data warehouse. Next, a "view" on top of this warehouse is created for each department that needs to use the data. Finally, a spiral development methodology is used to develop each department view into a DDSS.

Although the Inmon approach is useful for managing the integration of structured data, it isn't generally applicable to modern KE system development. Modern KE systems typically operate on unstructured data that does not fit well into the ERD models. Furthermore, modern KE systems are usually undertaken as small specific projects, and do not fit well in to the top-down approach prescribed by the Inmon approach. Lastly, the Inmon approach does not provide sufficient support for two important aspects of modern KE system development: understanding the user needs for the system and implementing the software system for satisfying these needs.

Kimball Approach

The Kimball approach [Kimball and Ross, 2000] takes a bottom-up approach to developing DDSSs. In Kimball's approach, the development process starts with selecting a business processes that would benefit from a DDSS. Next, the data being collected is

examined from the perspective of each business process in order to determine how the data should be applied. A central tool for accomplishing this is the dimensional model which uses a matrix to correlate each aspect of the business process with specific type of data. The dimensional models are then used to create DDSSs for supporting each business process. Finally, the multiple DDSSs are combined using data bus in order to build the data warehouse.

The Kimball approach provides a number of useful tools for aligning DDSSs with existing work processes. In particular, the dimensional model helps developers better understand the relationship between work processes and types of data being collected. However, most of these tools do not work as well for modern KE system design where unstructured data makes it difficult to define how each piece of data is being used.

4 Approach

In this chapter we describe our approach for creating a new process methodology for Knowledge Engineering (KE) system development. We start by developing a general model for solving complex problems: the Integrated Meta-Representational Model (IMRM). We then describe how the IMRM can be used to create new methodologies for the design and development of technical artifacts, e.g. software-based KE systems. Finally, we show the application the IMRM to the problem of KE system development and explain the resulting process methodology.

4.1 Integrated Meta-Representational Model

The process of solving a complex problem starts with the definition of an initial state which represents the problem to be satisfied and ends with the realization of a goal state which represents the desired solution to the problem (e.g. satisfaction of the need). For many complex problems, such as KE system development, the realization of the goal state requires the creation of a technical artifact, e.g. a software system, for satisfying the need. In such cases, the problem solving process can be thought of in terms of designing and developing the appropriate artifact for realizing the goal state.

Recent work in cognitive neuroscience has shown: the human brain is a representational system; the human-brain solves problems by developing representations appropriate for solving the problem ([Revonsuo, 2009], [Metzinger, 2003]). Based on these two observations we have developed an Integrated Meta-Representational Model (shown in Figure 4.1) that specifies a natural sequence of representations for progressing from a problem's initial state through the design and development of the desired artifact for solving the problem. Each level of representation is an abstraction that addresses a particular aspect of the artifact and moves the development process towards the goal state.

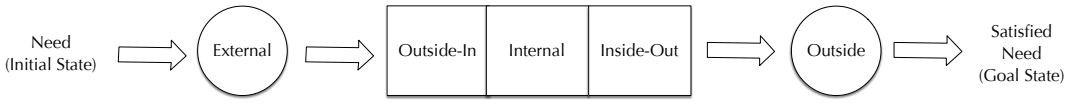

Figure 4.1: Integrated Meta-Representational Model

Given an initial state, the five representational levels of the IMRM are as follows:

- ***External***: represents the context for the initial state in terms of the way in which things are done at the present time. (Model of reality)
- ***Outside-In***: represents the transformation of the External into a set of requirements for the desired artifact. (Requirements for artifact)
- ***Internal***: represents the abstract functional specification of the *Outside-In* in a manner that enables the exploration and selection of form (design) for the desired artifact. (Function/Form design of artifact)
- ***Inside-Out***: represents the conversion of the *Internal* representation into a realizable design of the artifact. This level includes all the relevant domain knowledge particular to the design of the artifact. ("Domain" design of artifact)
- ***Outside***: represents the implementation process that realizes the *Inside-Out* representation into the actual artifact. (Development of the artifact)

4.2 Applying the Integrated Meta-Representational Model

The five levels in the Integrated Meta-Representational Model (IMRM) specify the minimal set of representations necessary for a comprehensive development process for technical artifacts. Addressing and resolving each level in the IMRM ensures that the resulting artifact will be complete and correct with respect to the needs specified by the initial state.

The IMRM can be applied in two ways. First, it can be used to analyze existing development methodologies for completeness, correctness, and consistency. Second, it provides a framework for creating new development methodologies that are minimal, complete, and correct. Since our goal is a development methodology for KE systems, we will focus on the application of the IMRM as framework for creating new development methodologies.

The process for applying the IMRM to create a new development methodology is as follows:

1. **Determine the subject matters being represented**: For each level of representation determine the appropriate subject matter that needs to be represented. For example, if we are developing a software system, the subject matter at the *Inside-Out* level of representation is the software design that specifies how the artifact will be constructed.

2. **Identify the necessary representational tools**: For each subject matter identify the appropriate set of domains and tools necessary for representation. For example, if our subject matter is the software design of a system then we will need tools from the domain of Software Engineering in order to represent the software architecture, data structures, control logic, etc.

3. **Integrate the selected tools**: The selected tools need to be integrated at two

levels. At the first level the individual tools for each level of representation must be integrated so that they can be used together to realize the subject matter. At the second level, the tools across the five levels of representation must be integrated so that the artifact is consistently represented throughout the development process.

4.3 Applying the IMRM to Create a Process Methodology for Knowledge Engineering system Development

In this section we describe application of the Integrated Meta-Representational Model (IMRM) to Knowledge Engineering (KE) system development.

Determine the subject matters being represented

The first step in applying the IMRM is to determine the subject matter that needs to be represented at each level of representation. The subject matters for KE system development are as follows:

- **External**: the subject matter is the current state of the organizational environment that the KE system will be operating in. This is where we represent the organizational context and current work process of the systems end-users in order to understand the key issues that the system will need to address.
- ***Outside-In***: the subject matter is the system from the perspective of the end-users. This is where we represent the user needs for the system in order to ensure that the developed system will be useful and attractive to the users. It is important to emphasize that the *Outside-In* level of representation is only concerned with the user perspective and does not consider the internal workings of the system.
- ***Internal***: the subject matter is the functional form of the system. This is where

we represent the functional specifications and explore multiple solution principles in order to select the best form based on the needs of the users and organization.

- ***Inside-Out***: the subject matter is the software design that can be implemented to realize the KE system. This is where we represent the overall architecture of the system and the solution-principles are formalized as data structures, algorithms, and control logic for the system.

- ***Outside***: the subject matter is the software implementation of the KE system. This is where represent the process for realizing the software design as executable code.

Identify the necessary representational tools

The next step in applying the IMRM is to identify the necessary representational tools for realizing the subject matters. In order to realize the subject matters for KE system development we will need representational tools from three domains: Knowledge Engineering (KE), Product Design (PD), and Software Engineering (SE). The KE domain provides the modelling tools for understanding the organizational context of system and work-process of the end-users. The PD domain provides formal tools for explicitly identifying user needs for the system, exploring different function realizations, and managing the inevitable trade-offs between a high-quality system that satisfies the end-user needs and minimizing development costs (time and money). Lastly, the SE domain provides the tools and techniques necessary for rapidly developing robust and reliable software systems. Table 4.1 shows subject matter and representational tools for each level of the IMRM.

Table 4.1: Methods and techniques for Knowledge Engineering system development

Level of Representation	Subject Matter	Methods and Techniques		
		KE	PD	SE
External (Level 1)	Organizational context and existing work process	CommonKADS Organization, Agent, and Task Models		
Outside-In (Level 2)	User needs		House of Quality	Use Case Diagrams
Internal (Level 3)	Functional specifications and solution-principles		Function Structure, Morphological Matrix, Utility Function	
Inside-Out (Level 4)	Software architecture and detailed design (data structures, algorithms, control logic)			UML Component and Class Diagrams
Outside (Level 5)	Software development			Incremental Development

Integrate the selected tools

The final step in applying the IMRM is to integrate the identified representational tools into a process methodology. The integration of the tools from Table 4.1 resulted in the creation of the Integrated Representation-Based Process Methodology (IRPM) for Knowledge Engineering system development. A comprehensive description of the IRPM is provided in Chapter 5.

5 Integrated Representation-Based Process Methodology

In this chapter we describe the integration of the methods and techniques from Knowledge Engineering, Product Design, and Software Engineering into an Integrated Representation-Based Process Methodology (IRPM) for Knowledge Engineering (KE) system development. We start with a high level overview of the different levels in the IRPM. We then proceed to provide a process for realizing the system at the each level of the IRPM. In order to concretely illustrate the methods and techniques at each level of representation, the network engineering problem—described in Section 2—is used as a running example throughout each step in the IRPM.

5.1 Overview of the IRPM

The five levels of the Integrated Representation-Based Process Methodology (IRPM) and their associated methods and connections are shown in Figure 5.1. In order to concretely illustrate how the methods and techniques at each level of representation the network engineering problem, described in Section 2, is used as a running example throughout each step in the IRPM.

At the *External* level of representation, we address the pre-processing necessary for creating a Knowledge Engineering (KE) system by creating models of the users' existing work process. These models are then used at the *Outside-In* level of representation to guide the process of defining the user needs for the KE system. At the *Internal* level of representation, these needs are used to create a design concept that defines the sub-functions and solution principles of the KE system. At the *Inside-Out* level of representation, the design concept is translated into the corresponding software design artifacts and then iteratively refined through multiple build and test cycles at the *Outside* level of representation.

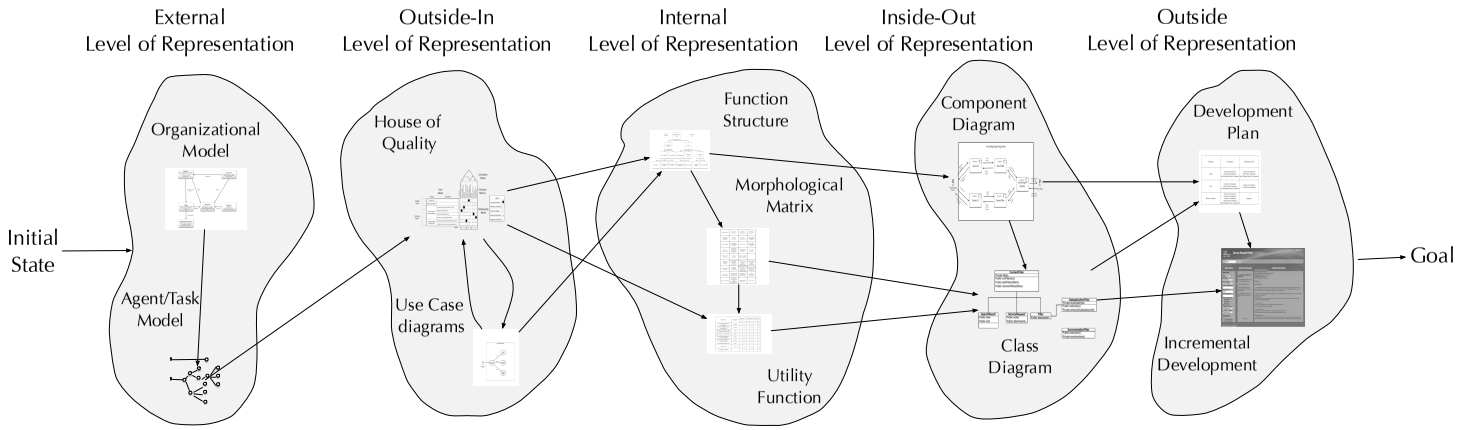

Figure 5.1: The five levels and associated methods of the Integrated Representation-Based Process Methodology for Knowledge Engineering system development

5.2 Level 1: External Representation

Knowledge Engineering (KE) systems do not operate in an organizational vacuum and must be integrated into the existing work processes and wider organization context in order to be valuable to the organization. The objective of the *External* level of representation is to capture these factors in a structured format that can be used to guide the development process in order to maximize the value of the system from the perspective of the organization. There are two important issues that will be addressed at the *External* level of representation: determining the overall organizational context for the system and capturing the work process in which the system will be operating.

The representation of the *External* is based on three models from the CommonKADS methodology [Schreiber, 1994]. The Organization model is used to represent the key organizational factors that come into play when developing KE systems. The Agent and Task models are used to represent the work process in the focus area and the structure of the individual tasks being performed.

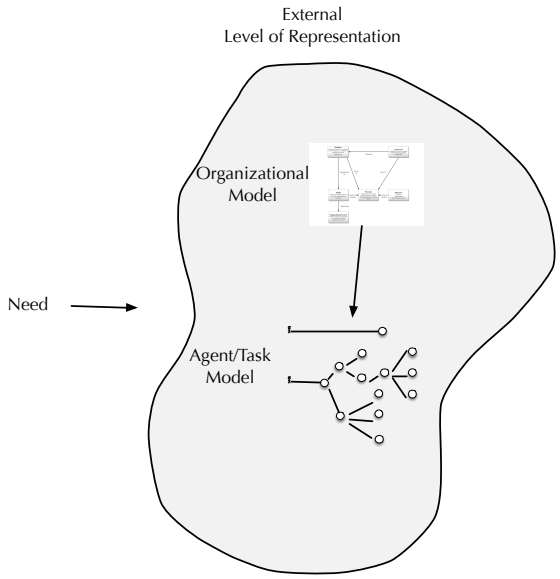

Figure 5.2: Methods and techniques at the *External* level of representation

The process for realizing the KE system at the *External* level of representation is as follows:

1. Build an **Organization model** to describe the organizational context and determine the focus area for the system.
2. Create an **Agent/Task Model** model of the work processes being performed in the focus area.

The *External* level of representation captures the current practices of the organization, determines what kind of KE system the organization needs, and how that system will fit into the existing work process. The *External* representation of the Service Request Portal (SRP) included one Organization model and one Agent/Task model. The Organization model identified that increasing efficiency of the problem-solution pair development process was an important and valuable organizational need. The Agent/Task uncovered the process bottlenecks—e.g. assessing the relevance of service requests—and helped define the user needs for the system. Together, the Organization and Agent/Task models ensured that the SRP was aligned with the organization's needs.

5.2.1 CommonKADS Organization Model

The development of a Knowledge Engineering (KE) system begins with an organizational need—e.g. improve the efficiency of a particular work process—that defines the initial state of the development process. However, it is unlikely that this need will be clearly articulated or that the organization will have a good idea of what needs to be done in order to satisfy the need. The CommonKADS Organization model [Schreiber, 1994] is a representation of the key organizational elements that come into play when developing a KE system and is used to refine the organizational need.

The Organization model for technical support organization where the Service Re-

quest Portal (SRP) was developed is shown in Figure 5.3. We have simplified the Organization model to include the key features that are most important to modern KE system development. These features include: organizational context, people, processes, resources, problems, and focus area. The organizational context describes the type of the organization where the KE will be operating. People are the users and supporting systems that will be interacting with the KE system. Processes are the work process that the KE system will be supporting. Resources are existing systems that are currently support the work process. Problems are issues that complicate or impact the work process. The focus area is the area of the work process that would benefit the most from a KE system.

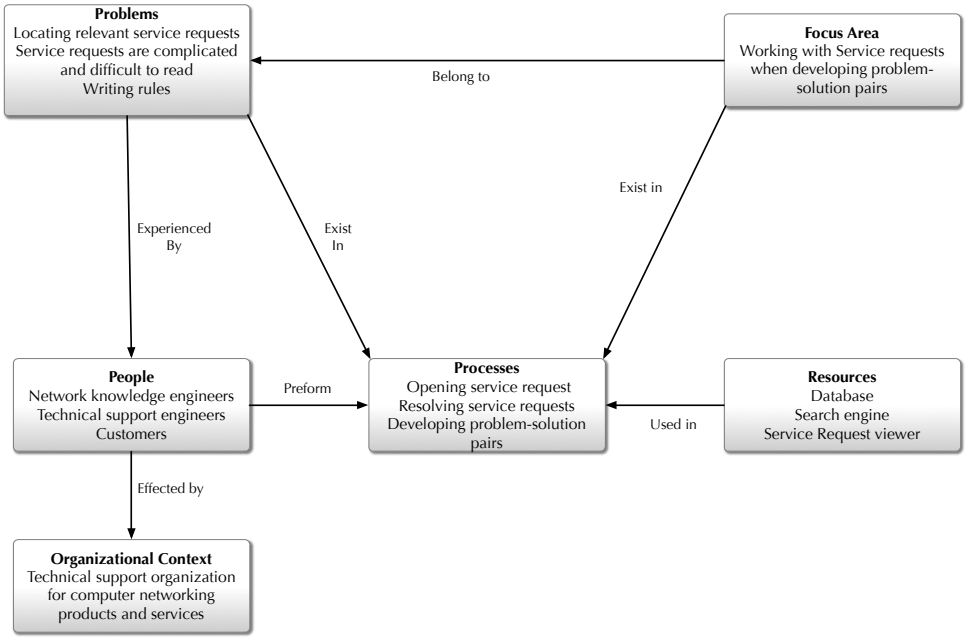

Figure 5.3: CommonKADS Organizational model for the technical support organization

In order to create the Organization model we start by describing the overall organizational context for the system. For example, the organizational context for the SRP was the support organization of a large networking company that provided technical support

and services to the customers that used the companys networking products. Next, we list the people involved in this context. In the support organization this included: customers, technical support engineers (TSEs), and network knowledge engineers (NKEs). Next, we capture the different work processes that these people perform. For example, customers open service requests, TSEs resolve these service requests, and NKEs mine the resolved service requests for problem-solution pairs. We then add the resources that are used by the people in order to perform the processes. For example, the NKEs use a search engine to query the database in order to find service requests. Next we identify the problems in these process. For example, one problem in the NKE work process is that it is difficult for the NKEs to locate relevant service requests. Finally we specify the focus area for the KE system, e.g. working with service requests when developing problem-solution pairs.

The process for constructing the Organization model is as follows:

1. Describe the organizational context for the KE system. Important features to consider are the mission, vision, goals of the organization, and strategy of the organization.
2. List the different processes being performed within organizational context.
3. Indicate the people who perform or are involved with the identified processes.
4. Describe the resources—search engines, databases, etc—that are being utilized to perform the processes.
5. Identify problems in the process based on interviews with knowledge workers and managers, brainstorming, shadowing, etc.
6. Work with the people to determine important problem as the focus area for the KE system.

The Organization model provides a concise high-level view of how the organization operates. Using this high-level view we can determine a focus area that constrains the development process and provides a rough idea of the KE system that we will need to build. For example, the Organization model for the SRP was used to determine that the focus area for the SRP would be the NKE When creating used guide the creation of the Agent/Task model that work process details and provides the starting point for creating the House of Quality that captures the user needs for the KE system.

5.2.2 CommonKADS Agent/Task Model

The Organization model provides the focus area for the Knowledge Engineering (KE) system. The next step is to drill-down into this focus area and capture the details of the work process that the system will be supporting. There are two tools in the CommonKADS methodology for doing this. The Agent model identifies the actors (users and external systems) who are involved in the processes in the focus area. The Task model is used to capture a task level decomposition of the different processes in the focus area.

We have combined the Agent and Task models into single integrated Agent/Task model. Figure 5.4 shows the Agent/Task model for the Network Knowledge Engineers' work process. The integrated Agent/Task model contains three different elements: tasks, agents, and relationships between agents and tasks. Agents represent people and systems that are involved in performing the processes within the focus area. Tasks are sub-parts or steps in a processes performed by agents. Relationships define which agents perform which tasks.

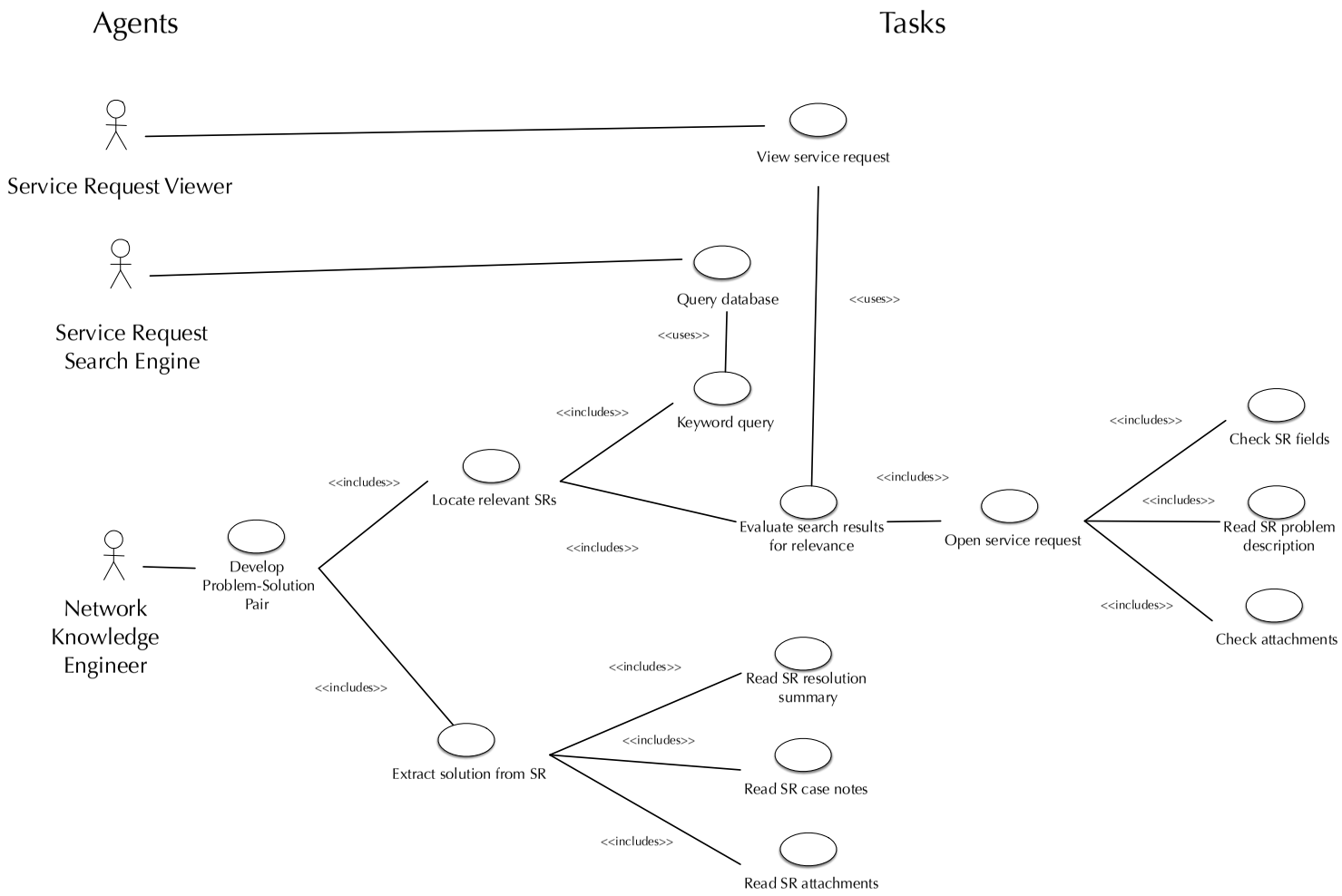

Figure 5.4: CommonKADS Agent/Task model of the network Knowledge engineer work process

In order to develop the Agent/Task model we first need to create a list of the different steps in the end-user work process. For example, the NKEs' work process for creating problem solution pairs for a particular problem of interest was as follows:

1. Formulate a set of keywords that described the problem of interest. Use the service request search engine to find a set of candidate (possibly relevant) service requests.
2. Evaluate each candidate service request for relevance.
   (a) Examine service request title for relevance
   (b) Open service requests with a relevant title using the service request viewer
   (c) Check the service request fields, e.g. software version, resolution code, to make sure they match the problem of interest
   (d) Read through service request problem description
   (e) Check to see if the service request has the right attachments to solve the problem of interest
3. Extract the solution from each relevant service request
   (a) Read the resolution summary to determine how the problem was resolved
   (b) Read case notes (emails, phone logs, etc) to get the action steps necessary to solve the problem
   (c) Read attachments and extract the relevant information

This work process is then converted into the graphical Agent/Task model. Each person or resource in the work process is represented as an agent. For example, the NKE work process included three different agents: NKEs that develop problem-solution pairs, the service request search engine that locates relevant service requests, and the service request viewer that displays the service requests. We next decompose the processes being performed by the agents into a set of tasks. For example, in order to develop

problem solution-pairs the NKEs used a search engine to locate candidate (possibly relevant) service requests. We then define the relationships between tasks. For example, developing problem-solution pairs includes requires searching for relevant service requests and reading service requests. The last step in creating the Agent/Task model is to connect agents to tasks that they perform. For example the NKE agent is connected to the task develop problem-solution pairs, the search engine agent is connected the task query database, and the viewer is connected with the task view service request.

The process for developing the Agent/Task model is as follows:

1. List the agents, people and resources, involved with the processes in the focus area defined in the Organization model.

2. Determine the specific tasks involved in the focus area processes. Use interviews and questionnaire to gather information about agent's current work process. Start by capturing high-level information, such as what they are trying to accomplish (the overall output of the work process). Next, determine the inputs and outputs to each task. Finally determine characteristics of each task such as relative difficulty, duration, frequency, etc.

3. Connect agents to the task that they perform.

4. Add the relationships among tasks. Tasks performed by a single actor are connected using the $<< includes >>$ relationships. Tasks across multiple actors are connected using the $<< uses >>$ relationship.

The Agent/Task model documents existing work process of the KE system's end-users in a graphical format that can be used to determine relationships between tasks and identify key bottlenecks in the current work process. For example, when developing the SRP the Agent/Task model exposed that the NKEs spent a considerable amount of time doing routine tasks, such as checking the fields of a service request when establishing

relevance. These insights were crucial in focusing on a simple and effective system that improved efficiency when extracting problem-solution pairs.

5.3 Level 2: Outside-In Representation

Once we have captured the organizational needs for the Knowledge Engineering system, the next step is to determine the user needs for the system. This takes us to the *Outside-In* level of representation, where the system is realized from the perspective of the end-users. The resulting set of user needs define the goal state for the development process and guide transition to the *Internal* level of representation. Issues that need to be addressed at this level include: capturing the user needs for the system, determining the technical metrics that can be used to evaluate how well the system satisfies the user needs, and defining the how the user will interact with system.

In order to realize the system at the *Outside-In* level of representation we have integrated two complimentary techniques for requirements elicitation: the House of Quality and UML Use Case diagrams. The House of Quality [Hauser and Clausing, 1988] is used to represent the user needs for the system. UML Use Case diagrams [Schach, 2008] are used to represent the interactions that the users will have in order to satisfy these needs.

In order to ensure that the system is tightly integrated into existing work processes, the realization of the system at the *Outside-In* level of representation must be driven by the Agent/Task model from the *External* level of representation. Figure 5.5 shows the connections between the Agent/Task model and the House of Quality, and Use Case diagrams. When creating the House of Quality, the Agent/Task model provides context necessary to identify user needs based on real problems in the existing work process. When creating the Use Case diagrams, the Agent/Task model provides the model necessary to ensure that the system will be compatible with current practices.

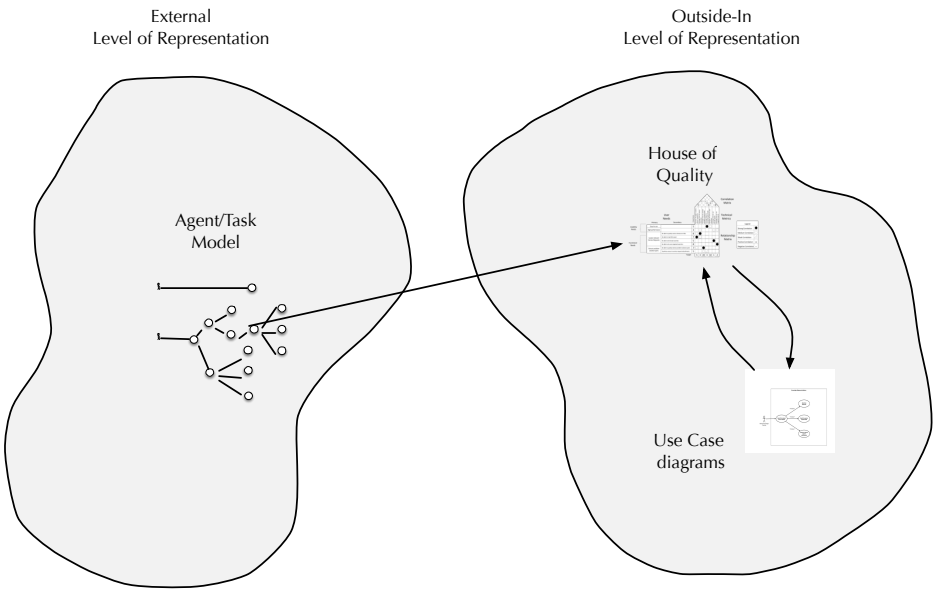

Figure 5.5: Methods and techniques at the *Outside-In* level of representation

The process for applying the House of Quality and Use Case diagrams to realize the system at the Outside-In level of representation is as follows:

1. Work with end-users to determine a prioritized set of user needs for the KE system based on the issues in the current work process. Define a corresponding set of technical metrics that can be used to measure the satisfaction of these needs. Correlate user needs to technical metrics in a **House of Quality** diagram.

2. Translate each functional user need into a goal that the user is trying to accomplish. Create a **UML Use Case diagram** to model how the users will interact with the system to accomplish that goal.

3. Iteratively refine the House of Quality and Use Case diagrams using feedback from the end-users.

The combination of the House of Quality and Use Case diagrams allows us to quickly obtain a high-quality set of user requirements for the KE system. The House of Quality

focuses the requirements process on the areas that are most important to the users. The Use Case diagrams then provide a more concrete realization of the user needs so that they can be evaluated for feasibility, utility, and difficulty. The iteration loop between the House of Quality and the Use Case diagrams minimizes errors in system requirements early in the development process where the cost of change is relatively inexpensive.

The House of Quality coupled with the Use Cases provide a user-centric framework for prioritization and decision making throughout the design and development process. For example, the realization of the Service Request Portal at the Outside-In level of representation consisted of one House of Quality and nine Use Case diagrams. The House of Quality was critical in defining the functions for the Function Structure, evaluating the different design concepts, and guiding the evaluation of the system during software development. The Use Case diagrams were critical in the defining the information flows for the Function Structure and implementing the UI components in the system's software architecture.

5.3.1 House of Quality

The House of Quality [Hauser and Clausing, 1988] captures the user needs for the system and translates them to a set of engineering targets (technical metrics) to be met by the new design. Although there are many variations of the House of Quality, the basic form [Hauser and Clausing, 1988] (shown in Figure 5.6) consists of five components: user needs, technical metrics, relationship matrix, and correlation matrix. User needs [Otto and Wood, 2000] are short descriptions of what the users (customers) desire from the system. Technical metrics [Ulrich and Eppinger, 1995] are precise, quantitative metrics that quantify how well the system satisfies the user needs. The relationship matrix establishes the connection between the user needs and the technical metrics. Finally, the correlation matrix captures the interdependencies between the technical metrics.

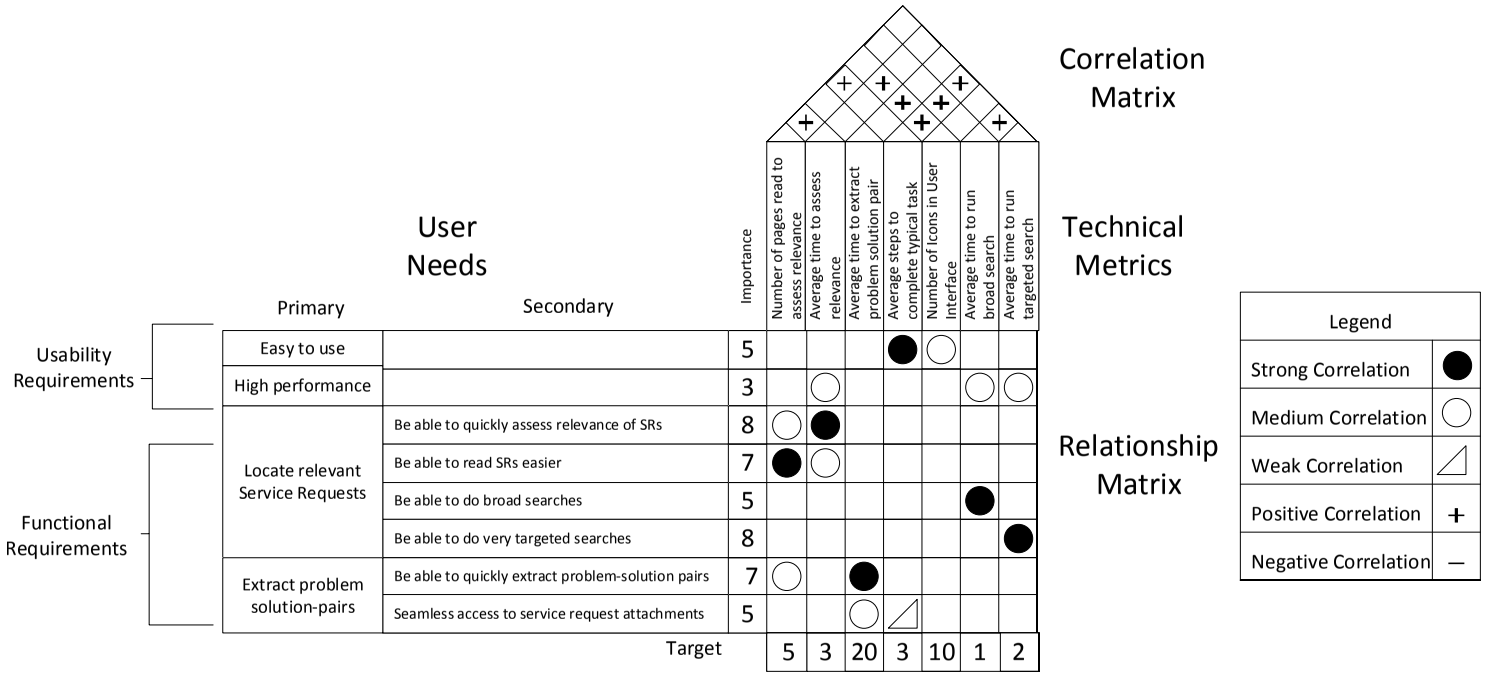

Figure 5.6: House of Quality of the user needs for the Service Request Portal

The first step in creating the House of Quality is to identify the user needs for the KE system. In general there are two different kinds of users needs to be considered: functional needs and usability needs. Functional needs define the functions that users need the system to perform and are identified by working with these users to identify the gaps and problems in their existing work process that could benefit from automation. For example, in order to locate relevant service requests the NKEs manually examined the fields (technology, sub-technology) of each service request. The NKEs wanted to be able to specify these field values as part of the search criteria, so that they did not have to manually filter search results. This user need was captured in the functional requirements be able to do very targeted searches. Usability needs are general characteristics, e.g. easy to use, that the users want the system to embody. Usability needs can be determined through interviews and questionnaires with the users.

Once we have determined the user needs for the system the next step is to define the technical metrics that can be used to measure these needs. For example, measuring how well the system satisfied the functional need be able to read service requests easier translated to the technical metrics: number of pages read to assess relevance and average time to assess relevance.

The last step in creating the House of Quality is to identify the relationships between user needs and technical metrics. For example, the user need be able to read service requests easier is strongly related to the technical metric number of pages read to assess relevance and number of pages read to assess relevance is positively correlated to the technical metric average time to assess relevance.

The process for building the House of Quality is as follows:

1. Identify the user needs for the system. The functional needs for the system are identified by working with end-users to determine problems in the work process captured by the Agent/Task model. Usability needs are identified by interviewing

users about what characteristics are important in order to make the system easy to use.

2. Work with the users (customers) to establish the relative importance of each need using a convenient scale (e.g. 1-10). Organize the user needs into a hierarchy of primary needs, secondary needs, and tertiary needs.

3. Make a list of technical metrics for measuring the factors that influence the satisfaction of the user needs. Use the Agent/Task model to understand the factors that influence the work process. Determine a target value for each technical metric that represents a significant improvement over the existing process.

4. Organize the user needs and technical metrics into a relationship matrix. Within the matrix identify the relationship between each user needs and technical metric as strong, moderate, weak, or no relationship. A strong relationship indicates that increasing or decreasing the technical metric will affect the user need; no relationship means that the user need and technical metric are independent of each other and changing the technical metric will have no effect on the user need.

5. Correlate the technical metrics to each other using a convenient scale (e.g. positive or negative correlation). A positive correlation means that the technical metrics increase/decrease together, while a negative correlation means that the technical metrics increase/decrease opposite to each other. The result is the correlation matrix.

6. Combine the user needs, technical metrics, relationship matrix, correlation matrix into a House of Quality. The relationship matrix occupies the center of the House of Quality, with the user needs are to the left and the technical metrics are above. The correlation matrix is positioned on top of the technical metrics. The target values for the technical metrics are positioned below the relationship matrix.

The House of Quality summarizes the user needs for the KE system in a concise

and well organized form that allows them be easily referenced throughout development process. At the *Outside-In* level of representation, the House of Quality ensures that the goals in the Use Cases are connected to real user needs for the system. At the *Internal* level of representation, the House of Quality is used to drive the creation of the Function Structure and the Utility Function. At the Outside level of representation, the House of Quality provides the technical metrics that can be used to evaluate the software with respect how well it satisfies the user needs. By properly emphasizing the user needs throughout the development process, the House of Quality improves the quality and cost effectiveness of the system because finite resources are focused on the features and functionality that are important to the users.

5.3.2 UML Use Case Diagram

The UML Use Case diagram [Schach, 2008] is a black-box representation of a particular aspect of the systems functionality from the perspective of the user. Figure 5.7, below, shows the Use Case diagram for how the Network Knowledge Engineers (NKEs) would interact with the Service Request Portal (SRP) in order to locate relevant service requests. The Use Case contains of four elements: actors that interact with the system, the software system, interactions between the actors and the software system, and the relationships between interactions. Actors represent any entity that interacts with the system and are drawn as stick figures. Interactions describe how these actors can interact with the software application and are drawn as ovals. Relationships describe the connections between interactions and are drawn using lines.

Each functional user need in the House of Quality will require a Use Case to capture how the user will interact with the system in order to satisfy that need. In order to create a Use Case diagram for a functional need we first transform the user need into a goal that the user trying to accomplish using the system. Next we will need to identify the different actors involved in that goal. There are two different kinds of actors: primary

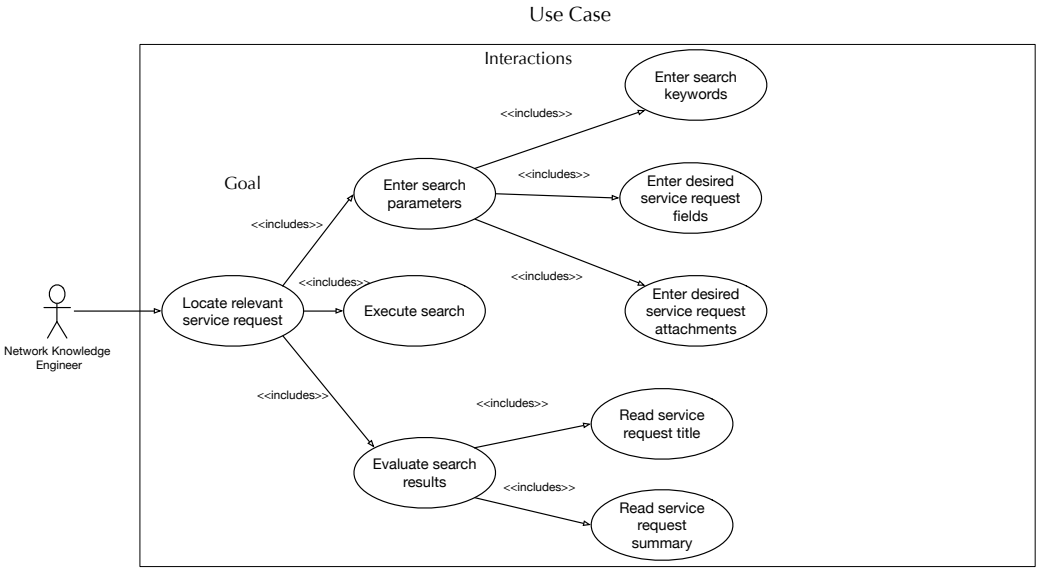

Figure 5.7: Use Case diagram for locating relevant service requests

and secondary. Primary actors represent the users that require the assistance of the system, while secondary actors represent external systems (search engines, databases, etc.) that the system uses. Finally, there is well-defined structure user interactions with the system. For example, the user will need to enter the search keywords before entering the service request fields. Use Case diagrams use relationships to capture this structure. Each relationship describes the nature of a connection between two interactions. There are two kinds of relationships that are of interest when designing KE systems: dependencies, and variations. Dependencies are where one interaction includes the other interaction, and are labeled with the $<< includes >>$ relationship. Variations, are where one interaction is a special case of another interaction, and are labelled with the $<< extends >>$ relationship.

The process for creating the Use Case for a functional user need is as follows:

1. Translate the user need into a goal that the user is trying to accomplish. Draw a box around the goal in order to represent the system boundary.

2. Add a primary actor to represent each distinct group of users (e.g. NKEs) and add a secondary actor to represent each external system (e.g. the search engine) that interacts with the KE system. Place primary actors to the left of the system boundary and secondary actors to the right.

3. Add a relationship between the primary actor and this goal.

4. Work with the users to add interactions for achieving the goal. Use the Agent/Task model to ensure that these interaction are compatible with the current work process.

5. Add the relationships between the interactions to the Use Case diagram. Specify each relationship in the Use Case diagram as either a functional dependency (<< *includes* >>) or variations (<< *extends* >>).

The application of this process to the House Quality will yield a set of Use Case diagrams, each describing a goal oriented set of interactions that the user can have with the KE system. For example, the translation of the SRP House of Quality resulted in the creation of nine Use Case diagrams. Together, these Use Cases provide a interaction based representation of the system that can be used refine the user needs and their respective priorities in the House of Quality. The interactions captured in the Use Case diagrams are also useful when defining the functional specification of the system at the *Internal* level of representation. Use cases ensure that the development team and the users are clear on how the user needs—captured in the House of Quality—will translate into system functionality.

5.3.3 Iterative Refinement

In the process of creating the Use Case diagrams it is likely that the user needs and their respective priorities will change. Users will discover new needs that were not cap-

tured in the House of Quality, or find that their previous prioritizations were incorrect. Therefore, it is typically necessary to iteratively refine the user needs in the House of Quality after creating the Use Cases.

The first step in refining the House of Quality and Use Case diagrams is to have the users review the Use Case diagrams. Based on their feedback there are three different types of corrections that will need to be made to the House of Quality. First, the Use Cases could have brought out an important user need that was not captured in the House of Quality. Second, the Use Cases could have exposed a user need in the House of Quality that is not necessary. Third, the Use Cases could have identified a change that will need to be made to the priorities of the user needs. Each of these corrections will need to be reflected in the House of Quality.

The process for refining the House of Quality and Use Case diagrams is as follows:

1. Have the users review the Use Cases and give feedback on what corrections need to be made.
2. Adjust the House of Quality. Add any new user needs that were discovered, remove unnecessary user needs, and adjust priorities.
3. Update the Use Case diagrams to reflect the changes to the House of Quality.
4. Repeat steps 1-3 until the House of Quality and Use Case diagrams are relatively stable and do not change between iterations.

Errors in the requirements phase are generally the most costly errors to fix and have prolonged effects on the overall user satisfaction with the system. The iteration loop between the House of Quality and the Use Case diagrams minimizes the number of the errors in system requirements early in the development process where the cost of change is relatively inexpensive.

5.4 Level 3: Internal Representation

Once we have captured the user needs for the system, the next step is to translate these user needs into functions and solution-principles for the Knowledge Engineering (KE) system. We refer to this process as the realization of the system at the *Internal* level of representation. Issues that will be addressed at this level include: defining the functional specification of the system, exploring different function realizations (solution-principles), and managing the inevitable trade-offs between system quality and cost.

In order to realize the system at the *Internal* level of representation we have drawn upon three Product Design techniques. The Function Structure [Pahl and Beitz, 1996] is used to represent the functions and sub-functions that the KE system will need to perform. The Morphological Matrix [Pahl and Beitz, 1996] is used to represent the space of feasible realizations for the system's Function Structure. The Utility Function [Cross, 1998] is used to represent the selection process for choosing the design concept that best satisfies both the user and organizational needs.

The realization of the system at the *Internal* level of representation must be driven and shaped by the user and organizational needs for the system. Figure 5.8 shows the interconnections between the Outside-In and *Internal* levels of representation. The Function Structure transforms the House of Quality and Use Case diagrams into the functional specifications for the system. The Utility Function transforms the House of Quality into a set of weighted selection criteria for evaluating how well the generated design concepts satisfy the user needs.

The process for applying these methods to realize the system at the level of the *Internal* representation is as follows:

1. Construct a **Function Structure** of the functions and sub-functions that the system must perform in order to satisfy the user needs specified in the House of

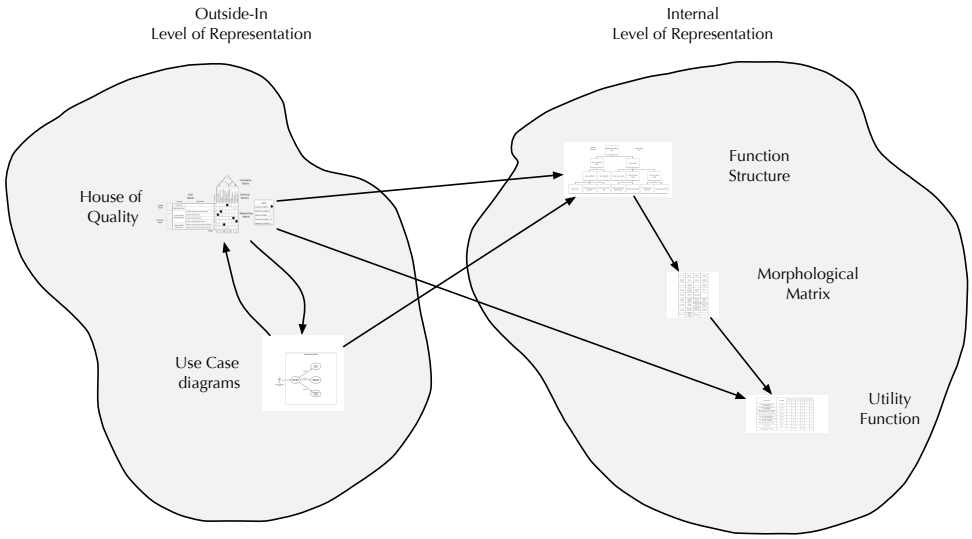

Figure 5.8: Methods and tools at the *Internal* level of representation

Quality.

2. Build a **Morphological Matrix** (MM) to capture the space of feasible realizations (solution-principles) for the Function Structure. Use the MM to generate several alternative design concepts for the KE system.

3. Develop a set of weighted objectives for the system, based the user and organizational needs for the system. Use a **Utility Function**, constructed from these objectives, to assess the alternative design concepts. Select the best design concept for further development.

The realization of the *Internal* level of representation provides a complete conceptual design for the KE system. This conceptual design is then translated into a software design at the *Inside-Out* level of representation. For example, the realization of the Service Request Portal (SRP) at the *Internal* level of representation included: one Function Structure, one Morphological Matrix, three alternative design concepts, and one Utility Function. The Function Structure was used to define the software architecture for the

SRP and selected design concept was used to guide the detailed software design. The Morphological Matrix used to explore the possible solution space and generate three alternative design concepts for the system. The Utility Function was used to select the "best" design concept based on the user and organizational needs.

5.4.1 Function Structure

The Function Structure [Pahl and Beitz, 1996] is a representation of the functions a system performs on a set of inputs in order to obtain a set of outputs. Figure 5.9 shows the Function Structure for functions and sub-functions of the Service Request Portal. A Function Structure is organized as a hierarchy, of a primary decomposed into a series of increasingly detailed sub-functions. The basic construct at each level is a function represented as a black box that takes a set of inputs and transforms them into a set of outputs.

The first step in creating the Function Structure is to define the primary function of the system. This main function must be determined from the House of Quality in order to ensure that the system satisfies the user needs. For example, the two primary functional needs captured in the House of Quality were: locating relevant service requests and extracting problem-solution pairs. These needs were combined in order to get the to the main function: Facilitate searching for relevant service requests and extracting problem-solution pairs. The information flow (inputs and outputs) for this main function comes from the overall interactions with the system that are specified in the Use Case diagrams.

The main function is then decomposed into a set of primary sub-functions. These primary sub-functions should roughly correspond to the primary functional user needs specified in the House of Quality. For example, the two primary user needs in the House of Quality each translated to a primary sub-function. This process continues until each

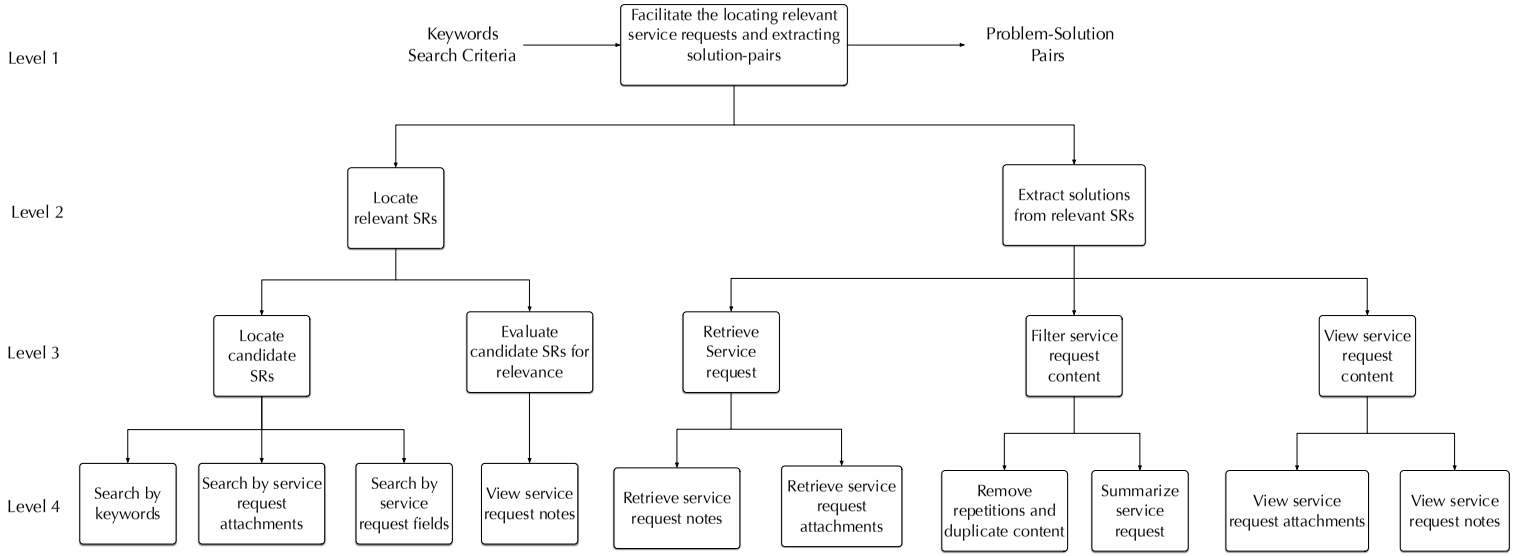

Figure 5.9: Function Structure for the Service Request Portal

sub-function is atomic and easy to realize. For example the Function Structure for the Service Request Portal had four levels. As a general rule each sub-function should be decomposed until it operates on a single input and output.

The process for creating the Function Structure is as follows:

1. Define the primary function for the KE system based on the functional user needs in the House of Quality. Use the Use Case diagrams to determine the information flow—inputs and outputs—for the primary function. This is the first level of the Function Structure.

2. Decompose the primary function into a set of sub-functions. Each sub-function should correspond to one primary functional need in the House of Quality. This is the second level of the Function Structure.

3. Continue this decomposition process until each sub-function is simple enough to realize. Each level of the Function Structure should correspond to a level in the user needs hierarchy, i.e. the third level secondary sub-functions correspond to the secondary functional need in the House of Quality.

The Function Structure translates the high-level user needs into a precise functional specifications for the KE system. These functional specifications provide the basis for the system design and are used throughout the *Internal* and *Inside-Out* levels of representation. At the *Internal* level of representation, the Function Structure provides the scaffolding of the Morphological Matrix and enables the generation of several alternative design concepts for the system. At the *Inside-Out* level of representation the Function Structure provides the functional software components that are used by the Component diagram to define overall software architecture of the system.

Function Structures have two key advantages over a written functional specification document. First and foremost, the hierarchical structure of the Function Structure

allows for clear visualization of the relationships and information flows between sub-functions and make it easier to the ensure that the design is functionally complete. Second, the Function Structure is abstract and clearly separates form from function. This separation enables (facilitates) the generation of a several feasible alternative design concept realizations and reduces the chance of preconceived solutions that do not satisfy real user (customer) needs.

5.4.2 Morphological Matrix

Once we have a functional specification for the desired system, the next step is to determine how these functions will be implemented. The Morphological Matrix [Pahl and Beitz, 1996] technique captures the potential solution space for the functional specification and provides a structured approach to generating alternative design concepts for the system. The Morphological Matrix that was used to develop the three design concepts for the Service Request Portal (SRP) is shown in Figure 5.10. The left-hand column of the matrix contains the systems functional specifications. The potential solution-principles or realizations for each sub-function are then listed in the right-hand columns. Design concepts can then be generated by selecting a solution-principles for each sub-function.

In order to create the Morphological Matrix we need to explore the space of feasible solution-principles for each sub-function defined in the Function Structure. For example, the Morphological Matrix in Figure 5.10 included different three solution-principles for the sub-function remove duplicate content: a strict hash function to detect duplicate paragraphs, a fuzzy hash function based off the similarity of the paragraphs, and a classifier to detect characteristics—e.g. $>$ — of duplicate paragraphs.

The process for creating the Morphological Matrix and using it to generate several design concepts for the KE system is as follows:

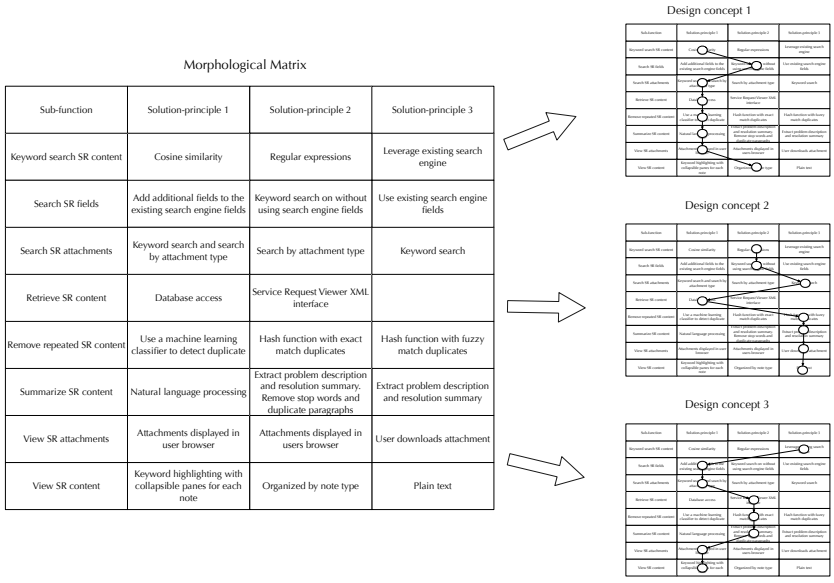

Figure 5.10: Morphological Matrix and three alternative design concepts for the Service Request Portal

1. Generate several (2-5) solution-principles for realizing each terminal (bottom-level) sub-function in the Function Structure.

2. Organize the terminal sub-functions and their corresponding solution-principles into the Morphological Matrix–where the sub-functions are the matrix rows and the solution-principles are the matrix columns.

3. Create several (2-3) copies of the Morphological Matrix. Combine compatible solution-principles in each Morphological Matrix to generate a unique alternative design concept for the KE system.

The Morphological Matrix technique yields a wide range of possible approaches for building the KE system. For example, the Morphological Matrix shown in Figure 5.10 was used to generate three very different design concepts. The first design concept used information retrieval techniques such as natural language processing for realizing the functional specification. The second design concept used a simple solution based

on regular expression matching. The third design concept was a hybrid approach that combined existing tools, e.g. search engine, with relatively simple enhancements. These alternative design concepts are then evaluated using a Utility Function in order to select one for further development.

The degree to which the developed system satisfies the needs of the users (customers) and organization largely depends on the quality of the underlying design concept. The Morphological Matrix allows the systematic generational a wide range of design concepts and therefore maximizes the probability of generating the best design with respect to these needs. For example, simpler designs, such as the second and third design concepts described above, might not have been considered if we had generated only a single design concept. However these designs ended up having a higher utility then the first design concept because they were less expensive to develop.

5.4.3 Utility Function

The Morphological Matrix allows for multiple feasible design concepts to be generated for the Knowledge Engineering (KE) system. However, it is not generally possible (due to cost and time constraints) to develop more than one design concept. Therefore, the last step in the *Internal* level of representation is to select the best design concept to develop.

The Utility Function [Cross, 1998] is a mathematical tool for assessing of the usefulness of the design concepts with respect to a set of weighted objectives. Figure 5.11 shows the Utility Function that was used to select the design concept for the Service Request Portal (SRP). The Utility Function assigns numerical weights to objectives and numerical scores to the design concepts measured against the objectives. The weighted scores ($numericalweightXnumericalscore$) are then summed in order to compute a cumulative or overall utility for each design concept. This cumulative utility can then

be used to compare the design concepts in order to select the best design for the KE system.

Objective	Weight	Concept 1		Concept 2		Concept 3	
		Score	Utility	Score	Utility	Score	Utility
Provide broad search capabilities	0.042	8	0.336	4	0.168	7	0.294
Provide targeted search capabilities	0.126	6	0.756	3	0.378	7	0.882
Be able to quickly assess the relevance of an SR	0.21	7	1.47	4	0.84	6	1.26
Easy access to attachments	0.044	4	0.176	2	0.088	7	0.308
Easy to find problem in service request	0.072	7	0.504	3	0.216	5	0.36
Easy to find resolution in service request	0.072	7	0.504	3	0.216	5	0.36
Fast development time	0.28	3	O.84	8	2.24	6	1.68
Low cost	0.12	3	0.36	4	0.48	7	0.84
Cumulative Utility			4.946		4.626		5.984

Figure 5.11: Utility Function for assessing the three Service Request Portal design concepts

In order to create the Utility Function we need to define a set of objectives that can be used to score the design concepts. There are typically two primary objectives when developing a KE system: high-quality, and low-cost (time and money). The quality objectives should be based on the user needs in the House of Quality. The cost objectives should capture the organizational needs for the system. Next we assign each objective a weight according its importance relative to the other objectives. The weights of the quality objectives are determined by the relative importance of each user

need. The weights for the other objectives, e.g. cost, should be assigned while working in collaboration with users, project stakeholders, and the software developers who will be implementing the system. Each design concept is assigned a score for each objective based on how well the concept satisfies that objective. For example, the design concept 3 used simpler (easier to implement) solution-principles than design concept 1 and, therefore, received a higher score (0.84 vs 0.12) for the objective low cost. Finally we calculate a cumulative utility for each design concept by summing the weighted scores.

The process for constructing and applying the Utility Function is as follows:

1. Generate a hierarchy of objectives for the KE system.

2. Assign each objective in the hierarchy a relative weight according to how important it is relative to the other objectives. The relative weights should be assigned so that relative weights of every objectives immediate descendants sum to 1.

3. Determine the absolute weight of each objective by multiplying the objectives relative weight by the absolute weight of its parent. The absolute weights of all the objectives at the same level must sum to 1. Therefore the relative and absolute weights of the root of the tree are both 1.

4. Select the bottom level (terminal) objectives and their absolute weights from the objective and arrange them into a table with the design concepts. The objectives go in the table rows, and the design concepts go in the table columns.

5. Assign each design concept a score for each objective , using a convenient scale (e.g. 1-10), based on how well the objective is satisfied.

6. Calculate the utility scores for each design concept by multiplying the objectives scores by their corresponding weights. Compute a cumulative utility for each design concept by summing the utility scores.

All design processes involve some form of concept selection where decisions are made about how the system will be implemented. Typically, these decisions are made informally and are therefore subject the individual biases of the design team. The Utility Function provides a structured approach to concept selection that allows inevitable trade-offs between quality and cost (time and money) to be objectively balanced while the concepts are still relatively abstract. As a result, the Utility Function increases the probability of selecting a design concept that best satisfies the user and organizational needs for the KE system. In addition, the Utility Function also provides a transparent documentation of the concept selection process that clearly shows the selection criteria that were used and how the design concepts were scored.

5.5 Level 4: Inside-Out Representation

The *Inside-Out* level of representation is where the selected design concept is translated into the software design for the Knowledge Engineering (KE) system. The two key issues that need to be addressed at this level of representation are defining the software architecture of the system and transforming the design concept's solution-principles into a detailed software design.

In order to realize the system at the *Inside-Out* level of representation we have drawn two models from the Unified Modelling Language (UML) [Schach, 2008]. The UML Component diagram [Schach, 2008] is used to represent high-level architecture of the system in terms of components and their interactions. The UML Class diagram [Schach, 2008] is used to represent the implementation details of each component in terms of objects, methods, and attributes.

The purpose of *Inside-Out* level of representation is to accurately translate the design concept from the *Internal* level of representation into a software design that can be used to implement the KE system at the *Outside* level of representation. The two

connections between the *Internal* and *Inside-Out* levels are shown in Figure 5.12. The Component diagram transforms the hierarchical Function Structure into a component based architecture for the system. The Class diagram transforms the high-level solution principles specified in the Morphological Matrix into the data structures, algorithms, and control logic necessary for a detailed software design.

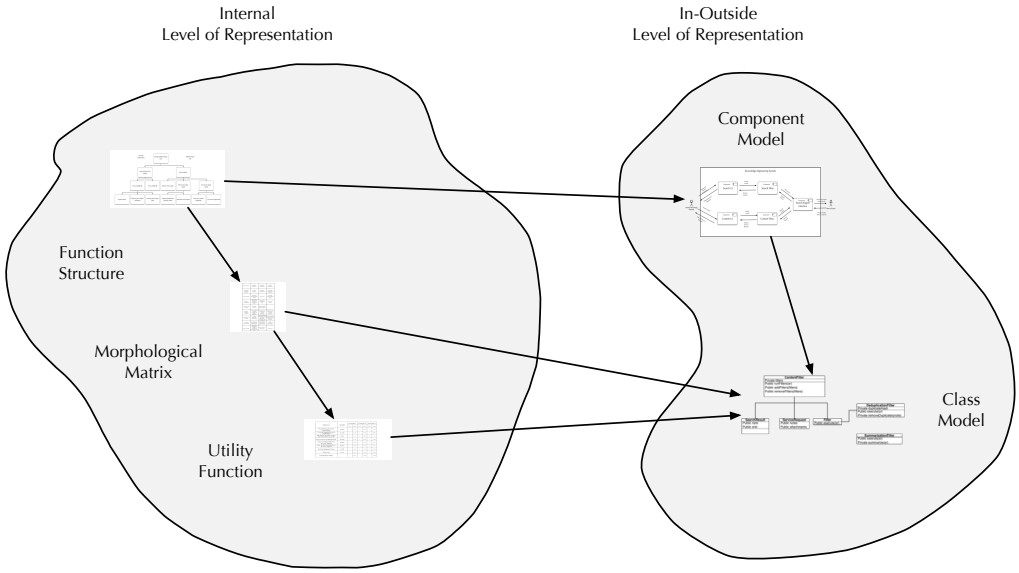

Figure 5.12: Methods and tools at the *Inside-Out* level of representation

The process for realizing the system at the level of the *Inside-Out* representation is as follows:

1. Map the sub-functions in the Function Structure to set of User Interface (UI) components, and the functional (backend) components required to implement the system based. Organize these components into a **Component diagram** that defines software architecture for the system.

2. Use **Class diagrams** to capture the implementation details of each component in the Component diagram based on solution-principles in the Morphological Matrix.

The realization of the system at the *Inside-Out* level of representation provides the software architecture and detailed design necessary to support software development at the *Outside* level of representation. For example, the realization of the Service Request Portal at the *Inside-Out* level of representation included: one Component diagram and over 50 different Class diagrams. The Component diagram was used create the software development plan for implementing the system and the Class diagrams were used to guide the actual software development (writing code).

5.5.1 UML Component Diagram

The UML Component diagram [Schach, 2008] represents the software architecture of a system in terms of components, actors that interact with components, and interactions between components. Figure 5.13 shows the Component diagram of the five components in the Service Request Portal (SRP) software architecture and their interactions. Each component is drawn as a rectangle with the component name listed at the top. Actors, representing users and external systems, are drawn using stick figures. Interactions between actors and components are drawn as arrows.

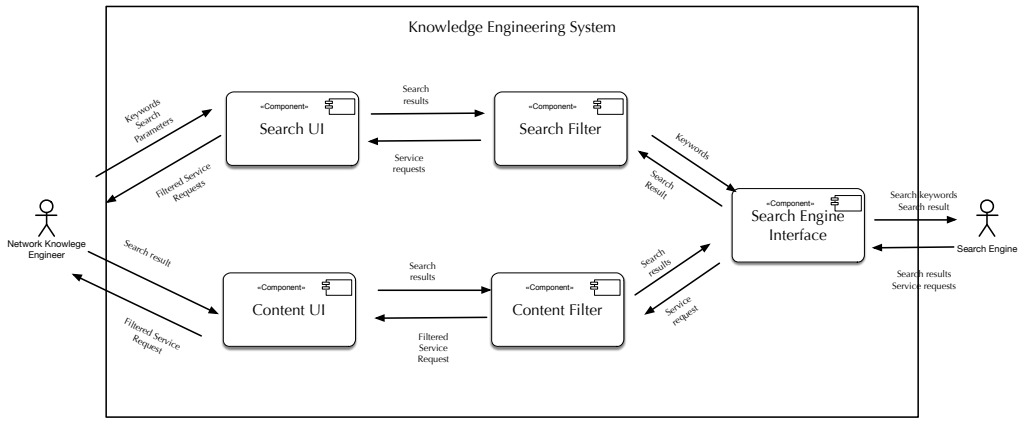

Figure 5.13: Component diagram of the Service Request Portal software architecture

The first step in constructing the Component diagram is to organize the systems functional representation (functions and sub-functions) into a set of software components. To do this we need to determine the appropriate level of the Function Structure at which to map sub-functions to components. The level should be chosen so that the mapping results in each component being as self-contained as possible with minimal coupling with other sub-functions outside of the component. For a small systems, e.g. the SRP, the appropriate mapping is generally the secondary sub-functions or the third level of Function Structure hierarchy. For large systems—with deeper Function Structure hierarchies—the fourth or even fifth level of the Function Structure is probably more appropriate.

Each sub-function at the selected level will map to either a user interface (UI) components that facilitates user interactions with the system or a functional components that implements the systems funtions. Sub-function that have an information flow with user inputs will map to a UI component while sub-functions that do not will map to a functional component.

Once we have defined the components, the next step is to specify the interactions between the UI components and the external environment. UI components are connected to the external environment through actors that interact with the system. These actors and their interactions with the system come from the Use Case diagrams. For example, the NKEs interact with the SearchUI component in order to search for relevant service requests.

The Functional components are then connected the UI components according to the information flows specified in the Function Structure. For example, the ContentFilter component takes a set a search result as input and returns a filtered service request as output.

The process for creating the Component diagram is as follows:

1. Determine the appropriate level of the Function Structure at which to map sub-functions to components. For each sub-function at this level determine if the sub-function should be mapped to a User Interface (UI) component that involves a user interaction or a functional component that does not involve user interaction. Place the functional components to the right of the UI components.

2. Add the system boundary by drawing a box around the functional and UI components.

3. Add an actor for each unique actor in the Use Case diagrams. Position primary actors (users) to left of the system boundary and secondary actors (databases, search engines, etc.) to the right.

4. Add the interactions between the actors and UI components. Label the information flows (inputs and outputs) involved in each interaction.

5. Add the interactions between the UI and the functional components based on the information flows in the Function Structure. Label the information flows (inputs and outputs) involved in each interaction.

The component diagram defines the software architecture for the KE system. At the *Inside-Out* level of representation this architecture is used to guide the construction of the Class diagrams that provide detailed design of the system. At the *Outside* level of representation this architecture is used create the development plan for implementing the KE system. Like any other complex structure, a software system must be built on a solid foundation or architecture. The Component diagram allows the system architecture to be defined in top-down manner so that important properties of good software architecture, such as modularity, simple interfaces, and information hiding, can be properly emphasized. A good software architecture will have less functional dependencies among components. This simplifies software development and allows multiple

components to be completed in parallel and will make the system easier to maintain and expand upon in the future.

5.5.2 UML Class Diagrams

Each software component, defined in the Component diagram, will require a Class diagram [Schach, 2008] to capture the implementation details necessary for software development. In general there are three different types of classes that are necessary when implementing software component: data structures, functional classes, and control classes. For example, the ContentFilter component required six classes—two data structure classes, two functional classes, and two control classes–to implement the transformation of a set of search results into filtered service requests. Figure 5.14 shows the ContentFilter component and its corresponding Class diagram. Each individual class is drawn as a rectangle, with the top portion of the rectangle containing the name of the class, the middle portion listing the class attributes, and the bottom portion listing the class functions.

Each software component in the Component diagram operates on an information flow. For example, the ContentFilter component takes a set of search results as input and returns a set of filtered service requests as output. The first step in transforming a component into a set of classes is to define the data structure for representing this information flow. Typically, we will need one data structure class for each distinct type of information flowing through the component. For example, the ContentFilter component required two data structures: the SearchResult class to represent the search results and ServiceRequest class to represent service requests.

The transformation of an input information flow into an output information flow requires a number of functional classes that contain the business logic of the component. Each Component will need functional classes that realize the solution-principles defined

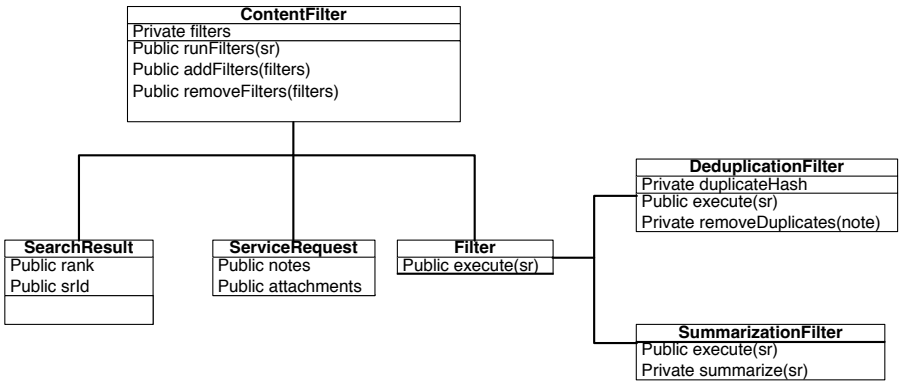

Figure 5.14: Class diagram for the Service Request Portal ContentFilter component

in the Morphological Matrix. For example, going from search results to filtered service requests required two functional classes: the DeduplicationFilter class for removing duplicated content and SummarizationFilter for summarizing service requests.

Each component will require coordination between several functional classes. In order to capture this logic we add control classes to receive inputs, trigger functions, and send outputs. For example, the ContentFilter and Filter classes specify the control logic for managing the retrieval and filtering of the service requests.

The last step in constructing the Class diagram is to define the relationships between the classes. There are three different types of relationships between classes: inheritance, aggregation, and association. Inheritance is when one class inherits (uses) the attributes and functions of the other class. For example, the DeDuplicationFilter and SummarizationFilter both inherit the execute() function from the Filter class. Aggregation is when one class is composed of multiple other classes. Association is any other type of relationship between two classes. For example the ContentFilter class is associated with

the SearchResult, ServiceRequest, and Filter classes.

The process for creating the Class diagram for a single software component is as follows:

1. Define a data structure class for each distinct input and output information flow to represent the components information inputs and outputs.
2. Add the functional classes to implement solution-principles from the Morphological Matrix for the transforming the input information into the output information.
3. Add the necessary control classes to receive inputs, trigger functions, and send outputs.
4. Specify the class relationships (inheritance, aggregation, association).

The application of this process to the Component diagram will yield one Class diagram for each functional and UI component. These class diagrams provide the blueprints for the realizing the system at the Outside level of representation. The advantage of Class diagrams is the specification of the systems implementation details before the software development. Class diagrams abstract the accidental complexity, such as syntax, bugs, etc., associated with writing code and allows us to focus on the essential complexity of the system.

5.6 Level 5: Outside Representation

The *Outside* level of representation is the actual implementation of the Knowledge Engineering (KE) system and is where the software design is realized as executable code. In addition to the ensuring that the code is functionally correct and free of bugs and defects, we also consider issues such as minimizing development costs, and testing the system with users.

In order to realize the system at the level of the outside representation we have adopted an Incremental Development approach [McConnell, 1996] where the system is incrementally developed through a set of build and test cycles. Each build and test cycle results in a prototype that delivers measurable benefits to the users.

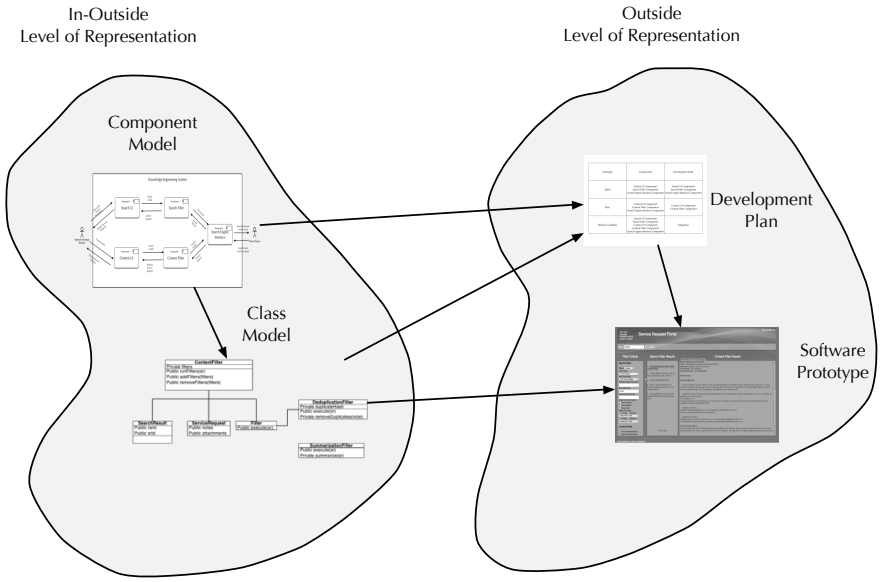

Figure 5.15: Methods and techniques at the *Outside* level of representation

The process for applying iterative software development is as follows:

1. Create a **development plan** [Schach, 2008] to guide the software development process. The development plan needs to include what prototypes will be developed, what development environment should be used for developing the prototypes, and how these prototypes should be tested with the end-users (customers).

2. Use rapid **build and test cycles** [McConnell, 1996] to iteratively implement the development plan, and elicit feedback from users.

The key benefit of using Incremental Development is the user interaction and feedback while developing the system. Frequent feedback ensures that the system will

sufficiently meet the users needs and increases adoption because the users are already familiar with the system. In parallel, Incremental Development also reduces the cost of developing the KE system because the user feedback prevents spending development resources (time and money) on unnecessary functionality or functionality that is already good enough to satisfy the user needs.

5.6.1 Software Development Plan

In order to efficiently manage the software development processes we require a software development plan to guide the build and test cycles. The software development plan addresses issues such as: how the software design should be segmented into prototypes, what development environment should be used for developing the prototypes, and how these prototypes should be tested with the end-users (customers). Figure 5.16 shows the three prototypes in the software development plan for the Service Request Portal.

Prototype	Components	Development Tasks
Alpha	Search UI Component Search Filter Component Search Engine Interface Component	Search UI Component Search Filter Component Search Engine Interface Component
Beta	Content UI Component Content Filter Component Search Engine Interface Component	Content UI Component Content Filter Component
Release Candidate	Search UI Component Search Filter Component Content UI Component Content Filter Component Search Engine Interface Component	Integration

Figure 5.16: Software development plan for the Service Request Portal

The first step in creating the development plan is to separate the software design

into a set of suitable prototypes. For example, the software development plan for the SRP included: alpha prototype with the search filtering functionality (Search UI, Search Filter, and Search Engine Interface), beta prototype with the content filtering functionality (Content UI, Content Filter, and Search Engine Interface), and release candidate (RC) prototype with the integration of these two sets of features (search filtering and content filtering) into complete Knowledge Engineering system.

The process for developing the software development plan is as follows:

1. Organize the components, defined in the Component diagram, into a set of development tasks. Small components should correspond to a single task, whereas large components can be broken up into several tasks according to their associated Class diagram.

2. Separate the development tasks into a series prototypes (short iterations), with each prototype being a user-testable partial implementation of the system. Use the House of Quality to determine the order in which the prototypes should be developed.

3. Define a suitable software environment (programming language, framework, etc.) in which to build and test the software prototypes. The software environment should take into consideration the experience of the development team, and should support rapid prototyping.

5.6.2 Build and Test Cycles

In order to develop the system with a high-level of user participation we have adopted an Incremental Development [Schach, 2008] approach where we implement a prototype and then test the software prototype with users to determine if it sufficiently satisfies the test criteria specified in the Software Development plan. If these criteria are satis-

fied, development proceeds to the next prototype (iteration) or deployment (if it is the last prototype). If the criteria are not sufficiently satisfied then the build-test cycle is repeated.

Each prototype specified in the software development plan will require one or more build and test cycles to implement. The process for performing a build and test cycle is as follows:

1. Implement the development tasks specified in the development plan. Use the Component diagram to understand the interfaces and interactions between tasks. Use the Class diagrams determine what classes need to be developed for each component.

2. Have the users actively use the prototype in their work process. Record feedback on user needs using a convenient scale (e.g. exceeds need, meets need, does not meet need). Record values for each technical metric in the House of Quality.

3. Use the user (customer) feedback and technical metrics to perform a gap analysis on the current prototype. If the user feedback and technical metrics are satisfactory then the prototype is finished, and the next step is to begin work on the next prototype. If the prototype is significantly below user expectations or technical metric targets, then return to Step 1 and perform another iteration.

The build and test process may result in a number of prototypes before we have a complete software system. For example, the SRP was developed in three iterations prototypes. An alpha prototype implemented the search filter functionality (Search UI, Search Filter, and Search Engine Interface). The beta prototype implemented content filter functionality (Content UI, Content Filter, and Search Engine Interface). The release candidate (RC) prototype integrated the search filter and content filter into a single software system. Each of these prototypes was actively used by the users in their

daily work process.

The key benefit of using Incremental Development is the user interaction and feedback while developing the system. Frequent feedback ensures that the system will sufficiently meet the users needs and increases adoption because the users are already familiar with the system. In parallel, Incremental Development also reduces the cost of developing the KE system because the user feedback prevents spending development resources (time and money) on unnecessary functionality or functionality that is already good enough to satisfy the user needs.

6 Results

In this chapter we describe the results of applying the Integrated Representation-Based Process Methodology to a real-world Knowledge Engineering problem in a large computer networking company. We start with a brief overview of the developed system, the Service Request Portal (SRP), and how it was used by the end-users. We then evaluate the SRP with respect to our overall objective: developing Knowledge Engineering systems that satisfy user needs and are high-value to the organization with respect to impact and cost.

6.1 Service Request Portal

The Service Request Portal (SRP) was used by Network Knowledge Engineers (NKEs) to locate relevant service requests and extract solutions developed by Technical Support Engineers. The NKEs interacted with the SRP through the web-based graphical user interface (GUI) shown in Figure 6.1. NKEs provided a set of search keywords and filter criteria in the left pane. The SRP then returned a set of service requests that match the keywords and satisfy the filter criteria in the middle pane. When the NKEs wanted to read a service request, the SRP retrieves the corresponding service request and presents an easy to read summary of the service request in the right pane. The summary contains important information such as the problem experienced by the customer, and steps taken by the technical support engineer to solve the problem. A detailed description of the SRP implementation is provided in Appendix A.

6.2 User Value of the Service Request Portal

In order to measure the value that the Service Request Portal (SRP) provided to the Network Knowledge Engineers (NKEs) who used the system, we evaluated the system

Figure 6.1: Service Request Portal Graphical User Interface

with respect to the user needs captured in the House of Quality. During each build and test cycle, the NKEs were asked to provide feedback for each of the user needs based on of their experience using the system. A summary of the user needs evaluation and feedback for the final build and test cycle is shown in Table 6.1.

In general, the NKEs found that the SRP significantly improved their productivity when workings with service requests. Two important user needs for the SRP were: be able to read service requests easier and be able to quickly assess the relevance of service requests". The SRP addressed these needs by automatically generating a high-level summary of the service request. In addition the NKEs found that the deduplication functionality was effective in removing the bulk of the irrelevant content and made the service requests much easier to read. The NKEs were also pleased that the SRP closely

Table 6.1: User feedback for the Service Request Portal

User Need	Evaluation	Feedback
Be able to do broad searches	Exceeds need	"The SRP helps us find information easily compared to the current search engine"
Be able to do very targeted searches	Meets need	
Be able to read service requests easier	Exceeds need	
Be able to quickly extract problem-solution pairs	Meets need	"The SRP is useful to extract the information quickly"
Seamless access to service request attachments	Meets need	
Easy to use	Exceeds need	"The Portal looks just amazing. Our team is excited to use it for rule writing."
High performance	Exceeds need	

integrated into their existing workflow, and did not have a significant learning curve.

6.3 Organizational Value of the Service Request Portal

In order to quantitatively evaluate the impact of the Service Request Portal (SRP) the Network Knowledge Engineers (NKEs) were benchmarked—using the technical metrics from the House of Quality—during the creation of nine problem-solution pairs ranging from relatively simple voltage alarms (R3) to complex hardware interface problems (R1). In this section we describe the results for the three technical metrics that best capture the impact of the SRP on the NKE work process. These technical metrics are as follows:

- average number of page read to assess relevance of a service request
- average time spent assessing relevance of a service request
- average time to extract the problem-solution pairs from the relevant service requests

The technical metric average number of page read to assess relevance of a service request, was used to measure how well the SRP satisfied the user need "be able to read service requests easier". Figure 6.2 shows the average number of page read to assess relevance of a service request for the nine problem-solution pairs R1-R9. We observed that the engineers went from reading 22 pages to reading 3 pages in order to assess relevance when using the SRP, suggesting that the summary was sufficient for establishing the relevance of a service request in most cases.

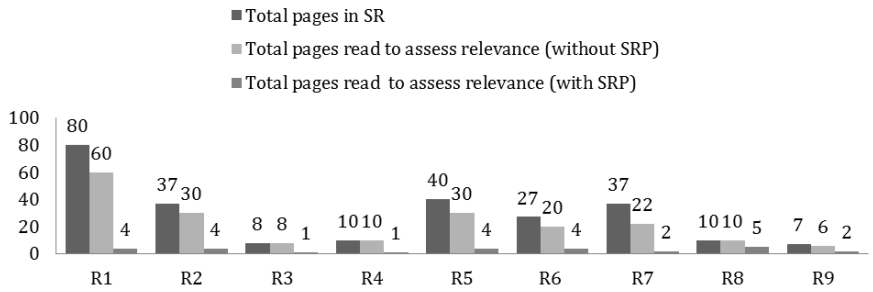

Figure 6.2: Average number of pages read to assess relevance of a service request

Reducing the number of pages the NKEs had to read translated to a 60% time savings when evaluating the relevance of service requests. Figure 6.3 shows the values recorded for the technical metric average time spent assessing relevance of a service request.

Once a service request was determined to be relevant, the NKEs performed a detailed read-through in order to extract the solution pair. Figure 6.4 shows the results for the technical metric average time to extract the problem-solution pairs from the relevant service requests. We believe that the time-savings can be attributed to the deduplication functionality that removed repeated information across threads. Other useful features such as keyword highlighting and allowing the users to view the attachment inside their

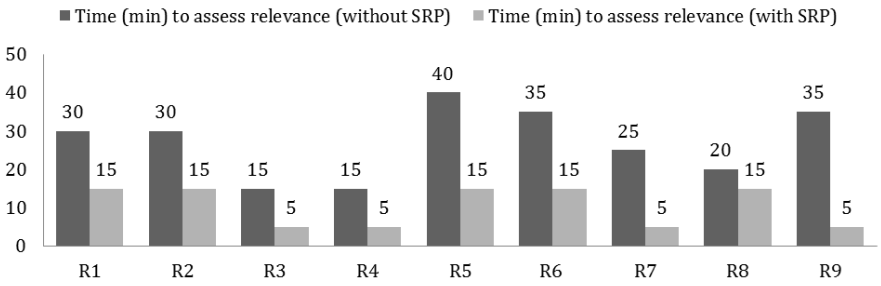

Figure 6.3: Average time to assess the relevance of a service request

web browser increased efficiency when working on longer service requests.

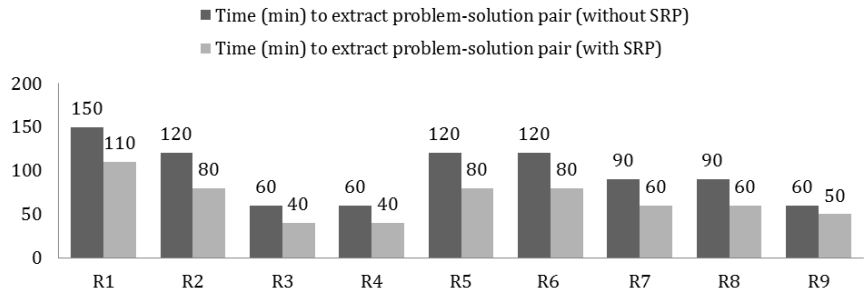

Figure 6.4: Average time to extract problem-solution pair from a service request

In order to measure the value of the SRP to the organization we compared the increase in NKEs productivity to the overall cost of developing the system. The SRP reduced the average time spent assessing the relevance of each service request by 60% (15 minutes) and time to extract the problem-solution pair by 30% (30 minutes). Since

NKEs read on average approximately ten service requests daily and extract problem-solution pairs from two of these service requests, we estimate, based on the technical metrics, that the SRP improved the daily productivity of the team of ten NKEs by 35 hours. The SRP was designed and developed by two student interns working over a six-month period (approximately 1000 man hours). Assuming that the interns' and NKEs time is equally valuable to the organization, the system represents a positive return on investment after one and a half months (43 days) of use.

7 Discussion

In this chapter we assess of the overall utility of the Integrated Representation-Based Process Methodology (IRPM). We first describe the impact of each domain to the development process. This then leads to a comparison with a . Finally, we describe possible simplifications for situations where it is either necessary or beneficial to simplify the IRPM.

7.1 Impact of Each Domain

The Integrated Representation-Based Process Methodology (IRPM) integrates methods and techniques from three distinct domains: Knowledge Engineering, Product Design, and Software Engineering. In this section we briefly discuss the impact that each domain had on the development process and describe what would have been lost without the methods from that domain.

7.1.1 Knowledge Engineering

The domain of Knowledge Engineering (KE) contributed the modelling tools necessary to realize the system at the *External* level of representation. The Organizational model documented the organizational context and the high-level processes occurring within this context. The Agent/Task model allowed for a creation of a detailed task-level decomposition of the end-users work process.

Without these KE tools the IRPM would lack a explicit model of the user work-process to guide the realization of the system at the *Outside-In*level of representation. As a consequence there is greater possibility of misunderstandings of the user needs when building the House of Quality and/or creating Use Cases for a system isn't practical given the end-users' work process. When developing the Service Request Portal the KE

tools allowed us to identify several "low hanging fruit" in the existing work process, such as allowing the NKEs to filter service requests by the meta-data fields, that could be easily automated. Without an explicit model of the user work process, we might have easily missed these high-value low-effort problems and focused our attention on problems that had lower value payoffs.

7.1.2 Product Design

The domain of Product Design (PD) contributed the conceptual design tools necessary to realize the system at the Outside-In and *Internal* levels of representation. At the *Outside-In*level of representation, the House of Quality helped guide the process creating Use Cases and ensured that every Use Case was related a real user need. At the *Internal* level of representation, the Function Structure, Morphological Matrix, and Utility Function methods enabled creation of a high-value design concept KE system.

The primary impact of the PD tools is at the *Internal* level of representation. Without the PD tools for *Internal* representation, the creative process of translating user needs into functional specifications, exploring the space of possible solution-principles, and selecting the best mix of solution-principles is largely dependent on the individual effort, skill, and experience of the designer. The use of Function Structures, Morphological Matrices, and Utility Functions, formalizes this process and moves it away from intuitive ("ad-hoc") methods to robust, repeatable methods that allow for a wide range of designs to be generated and evaluated. When developing the Service Request Portal we found that the systematic generation of the multiple designs enabled the discovery of a relatively simple design that satisfied user needs and was inexpensive (time and money) to implement.

7.1.3 Software Engineering

The domain of Software Engineering (SE) contributed the tools necessary to realize the system at the *Outside-In Inside-Out*, and Outside levels of representation. The UML Use Case diagram was critical in concretely describing the high-level user needs in the House of Quality at the Outside-In level of representation. The UML Component and Class diagrams allowed us to translate the abstract design concept into realizable software design at the *Inside-Out* level of representation. Finally, the Incremental development approach allowed for the KE system to be developed iteratively with user feedback at the *Outside* level of representation.

The primary impact of the SE tools is at the *Inside-Out* level of representation. Without a intermediate "domain" (software) design, the process of realizing the design concept is both inefficient and error-prone. Software design issues, such as modularity and information hiding, are addressed during software development, generally leading to an inefficient code and fix model where developers 1) write some code, 2) fix the problems in the code. When developing the Service Request Portal, the SE tools allowed us to efficiently translate the solution-principles specified in the design concept into a high-quality software architecture and detailed design. Addressing this translation during software development would have been significantly more difficult and likely would of resulted in a system that would difficult to maintain in the future.

7.2 Comparison to Analytics for Knowledge Engineering Approach

In the Introduction (Chapter 1 we described two aspects of Knowledge Engineering (KE) system development: Software Infrastructure for Knowledge Engineering, and Analytics for Knowledge Engineering. In this thesis we have focused on the Software Infrastructure for Knowledge Engineering aspects of the system. We now briefly compare it to an alternative approach that focuses on the Analytics for Knowledge Engineering aspects

of the system.

A typical "Analytics for Knowledge Engineering" approach is based on using Data Mining (DM) and Information Retrieval (IR) software toolkits to build the KE system. Two well-known DM/IR toolkits that support this approach are Lemur and Weka. The Lemur toolkit [Croft et al., 2009] provides search engine infrastructure–such as indexers, stemmers, natural language processers, etc.–that can be used to implement the information retrieval aspects of the KE system. The Weka toolkit [Witten and Frank, 2005] provides a library of machine learning algorithms, e.g. Support Vector Machines, and tools for data pre-processing that are useful for addressing the data mining aspects.

Although these toolkits and frameworks are useful in addressing the technical aspects of the system, they do not provide a structured approach that addresses the organizational context, user needs, business objectives for the system. Consequently, KE systems developed using this approach can provide low value to users and the organization because they don't address their needs and are poorly integrated into the existing work process. Furthermore, these systems often suffer from reliability issues and are difficult to maintain because they are pieced together from many components.

Comparison of the developed system with an Analytics for Knowledge Engineering based approach [Wang et al., 2010] to a similar Knowledge Engineering problem in the computer networking domain, shows that a Software Infrastructure for Analytics approach allowed us to achieve comparable results using significantly less complex analytical (Data Mining/Information Retrieval) components.

7.3 Simplifications to the IRPM

We recommend the formal methods described in Chapter 5 be used whenever possible. However there are situations when it is either beneficial or necessary to simplify the

application of these methods. For example, a time constraint might prohibit the use of the full IRPM. Similarly, if the Knowledge Engineering problem is not overly complex, simplifications can be used to reduce development time and costs. The recommended simplifications at each level of representation for these scenarios are shown in Table 7.3:

Table 7.1: Simplifications to the Integrated Representation-Based Process Methodology

Level of Representation	Method	Simplifications	Impact (Consequences)
External	Organizational Model	Just list the people and processes	Might not be focused on the right problem.
	Agent/Task Model	List tasks without creating a graphical model	Might miss important relationships between tasks
Inside-Out	House of Quality	List the user needs and technical metrics without creating the correlation matrices	More difficult to make tradeoffs during the development process
	Use Case Diagrams	Simplification is not recommended	
Internal	Function Structure	Reduce depth of the Function Structure. Omit information flows.	More difficult to come up with solution-principles when creating the Morphological Matrix.
	Morphological Matrix	Reduce the number of solution-principles explored for each sub-function.	Reduced chance of generating the best design based on the user and organizational needs
	Utility Function	Use un-weighted objectives	Loss of objectivity during design selection.
Outside-In	Component Diagram	Simplification is not recommended	
	Class Diagrams	Simplification is not recommended	
Outside	Software Development Plan	Simplification is not recommended	
	Incremental Development	Reduce the number of iterations	Less user feedback

8 Conclusions and Future Work

There are an increasing number of organizations attempting to develop software-based Knowledge Engineering (KE) systems to support the transformation massive amounts of data and information into useful knowledge that can be used to influence core business activities, e.g. product development, customer support, and marketing. However, the deployment of these systems often does not yield useful results, in particular because insufficient attention is spent addressing the needs of users that will be using the system.

In this thesis we have presented an Integrated Meta-Representation Model (IMRM) for structuring complex problem solving processes. The application of the IMRM to the Knowledge Engineering system development process resulted in an Integrated Representation-Based Process Methodology (IRPM) that combines methods and techniques from the domains of Knowledge Engineering, Product Design, and Software Engineering into a unified framework for developing Knowledge Engineering systems. Our novel contribution is the use of methods and techniques from Product Design in order to ensure that the developed system sufficiently addresses the users needs and represents the best value (quality and cost). We have demonstrated the effectiveness of the framework within the context of a simple but non-trivial Knowledge Engineering problem within the domain of computer networks.

The IRPM should prove to be valuable to the increasing number of organizations attempting to develop software-based KE Systems to support the transformation massive amounts of unstructured data and information into useful knowledge that can be used to influence core business activities, e.g. product development, customer support, and marketing. The IRPM provides a structured set of tools for resolving typical trade-off conflicts that arise in the design, development, and deployment of a Knowledge Engineering system. Based on our experiences, we have found that the framework can simplify the work in Knowledge and Software Engineering domains while substantially

increasing the quality and cost effectiveness of the system.

There are two general directions for future work: applying the IMRM to other complex problems, and refining the IRPM. Possible IRPM refinements include: expanding on the *Internal* level of representation to include a structured method selecting between more sophisticated Data Mining and Information Retrieval techniques and including other useful Software Engineering methods such as the UML Sequence diagram. The IRPM also needs to be further tested on wider range of Knowledge Engineering problems.

References

J. Angele, D. Fensel, D. Landes, S. Neubert, and R. Studer. Model-based and incremental knowledge engineering: The mike approach. *AIFIPP*, 1992.

A. Boehm, B. Jain. An initial theory of value-based software engineering. *Springer Verlag*, 2005.

B. Boehm. Value-based software engineering. *ACM Software Engineering Notes*, 2003a.

L. Boehm, B. Huang. Value-based software engineering: A case study. ieee computer. *IEEE Computer*, 2003b.

B. Croft, D. Metzler, and T. Stroham. *Search Engines: Information Retrieval in Practice*. Addison Wesley, 2009.

N. Cross. *Engineering Design: Strategies for Product Design*. Wiley, 1998.

J. Fox. *Quality Through Design: The Key to Successful Product Delivery*. Mcgraw-Hill, 1993.

J. Hauser and D. Clausing. The house of quality. *Harvard Business Review*, 1988.

W. Inmon. *Building the Data Warehouse*. Wiley, 2002.

M. Jackson. *System Development*. Prentice Hall, 1983.

Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison Wesley, 2001.

S. Kendall and M. Creen. *An Introduction to Knowledge Engineering*. Springer, 2007.

R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modelling*. Wiley, 2000.

S. McConnell. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, 1996.

T. Metzinger. *Being No-One*. Bradford, 2003.

K. Otto and K Wood. *Product Design: Techniques in Reverse Engineering and Product Design*. Prentice Hall, 2000.

G. Pahl and W. Beitz. *Engineering Design: A Systematic Approach*. Springer-Verlag, 1996.

A. Revonsuo. *Inner Presence: Consciousness as a Biological Phenomenon*. MIT Press, 2009.

S. Schach. *Object Oriented Software Engineering*. Prentice Hall, 2008.

B. Schreiber, G. Wielinga. Commonkads: a comprehensive methodology for kbs development. *IEEE Expert Systems*, 1994.

S. Spangler and J. Kreulen. *Mining the Talk: Unlocking the Business Value in Unstructured Information*. IBM Press, 2008.

K. Ulrich and S. Eppinger. *Product Design and Development*. Prentice Hall, 1995.

C. Wang, R. Akella, and S. Ramachandran. Hierarchical service analytics for improving productivity in an enterprise service center. *Conference on Information and Knowledge Management*, 2010.

I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

Appendices

A The Service Request Portal

In this section we describe the implementation of the Service Request Portal (SRP) Knowledge Engineering (KE) system. We start with a brief overview of the key functionality of the SRP and how it was used by the Network Knowledge Engineers. We then outline the high architecture of the SRP and describe its mechanisms for retrieving, processing, and displaying service requests. Lastly, we describe the set of technologies that were used to implement the SRP.

A.1 Service Request Portal Features

Service Request Portal (SRP), was used by Network Knowledge Engineers (NKEs) to locate relevant service requests and extract solutions developed by Technical Support Engineers. The graphical user interface (GUI) of the SRP is shown in Figure A.1. NKEs provide a set of search keywords and filter criteria in the left pane. The SRP then returns a set of service requests that match the keywords and satisfy the filter criteria in the middle pane. When the NKEs want to read a service request, the SRP retrieves the corresponding service request and presents an easy to read summary of the service request in the right pane. The summary contains important information such as problem experienced by the customer and steps taken by the technical support engineer to solve the problem.

A.2 System Architecture

The SRP can be divided into four core functionalities: user input, data retrieval, data processing and output results. In this section we will go through the high level organi-

Figure A.1: Service Request Portal User Interface

zation and structure of these core functions.

A.2.1 User Input

The SRP takes a set of user parameters that are used to generate the result set. There are two different kinds of parameters: parameters that are passed directly to search engine and parameters that are used additional constraints for the search result set.

A.2.2 Data Retrieval

The SRP uses service request data is contained within a Relational database. Within this database each service request is structured into two parts: metadata (SR ID, date,

contract ID, etc.) and a set of notes (emails, phone logs, etc.). There are two major systems that serve as the access points to service request data: a service request search engine, and a service request viewer.

- The service request search engine (SRSE) allows for keyword search across a text index of the service request data. The SRSE also supports several other search parameters such as searching by contract ID, technology, sub-technology, etc.
- The service request viewer (SRV) takes service request ID as input and outputs an HTML page containing the service request content. The SRSE links its results (using the service request ID) into the SRV. The SRV can be used also be used in an XML mode (the SRP uses this to build service requests data structures).

The Service Request Portal uses a set of "Retrieval Engines" to retrieve data from *External* sources. The Retrieval Engines share a set of base classes that provide the abstract interfaces for retrieving data. The actual logic specific to each *External* data source, is handled in the implementation classes that inherit from these base classes. Retrieval Engines operate upon a simple Request data structure object. The Request object contains two fields, parameters and data. The Parameters field with the information necessary to process the request. The data field is used by the Retrieval Engine to store the data resulting from the request.

The SRP retrieves data using two Retrieval Engines to pull data from the SRSE and the SRV. Both Retrieval Engines utilize a generic Retrieval Engine base classes and implement logic specific to each system. The SRSE Retrieval Engine is responsible for executing queries and scraping the service request IDs from the SRSE results page. The SRV Retrieval Engine is responsible for retrieving the XML data from service request viewer and parsing in order to build a ServiceRequest data structure.

A.2.3 Data Processing

The SRP processes data using a set of Filter Rules that are applied to ServiceRequest data structures. There are two different kinds of Filter Rules: Results Filter Rules for pruning search results and Content Filter Rules for pruning the content of individual ServiceRequests.

Results Filter Rules are applied against the search results set in order determine which results will be used in the SRP result set. Each Results Filter Rule operates on a ServiceRequest data structure and returns a Boolean value depending whether the ServiceRequest satisfies the Results Filter Rule criteria.

Content Filter Rules are applied against the SRP result set in order to modify the content of each result. Each Content Filter Rule operates on a ServiceRequest data structure and returns a modified copy. The SRP uses Content Filters for removing repetitions, cleaning up poor formatting, and generating summary content (first and last correspondence.).

A.2.4 Display Results

The SRP provides two different "views" on a ServiceRequest data structure: "Summary" and "Tidy". Each view provides a different level of detail for understanding the service request. The "Summary" view extracts out essential service request information and displays it in a concise easy to read format that can be used to quickly evaluate the relevance. The "Tidy" view provides a complete but cleaned up (deduplicated, fixed formatting) version of the service request. The purpose of the Tidy view is to cut down on the large amount of repetition and eyesore without losing any important information.

A.3 Technology Stack

In this section we briefly describe the technologies that are used to implement the Service Request Portal (SRP).

Ruby

The SRP is primarily implemented in the Ruby programming language. Ruby is a high level scripting language similar in syntax and functionality to other scripting language such as Python.

Ruby on Rails

Ruby on Rails is a framework for building web application using the Ruby programming language. Ruby on Rails employs the model view controller (MVC) architecture. The MVC structure enforces distinct separation between the business logic, interface, and system control. The models contain the business logic classes that are responsible for the core functionality. The views are the Graphical User Interface (GUI) where inputs are taken, and the results are shown. The controller is responsible for handling the flow of data between the models and views.

MySQL

MySQL is a relational database management system based on the Structure Query Language (SQL). The SRP uses MySQL to store persistent data. MySQL was selected because of its open source licensing, excellent documentation, and tight integration with Ruby on Rails.

BackgroundRB

BackgroundRB is an asynchronous job scheduler for Ruby on Rails. The BackgroundRB plugin consists of a number of workers and a BackgroundRB server to coordinate these workers. A worker consists of a class that contains methods that can be run asynchronously. These workers are instantiated in the main controller and then used to get a result set. The workers use the database to retrieve the request parameters and to write back results. Status information is written back using Memcached.

The SRP uses BackgroundRB to separate the long running jobs (retrieving and processing data) from the user interface. This prevents the user interface from blocking and becoming unresponsive while these long running jobs are waiting to complete.

Memcached

Memcached is a high performance RAM caching system that is used with Rails. The SRP uses Memcached to facilitate communications with running BackgroundRB workers. The BackgroundRB workers write status and results information to Memcached which can then be queried from Rails in order to provide the end user with status information.