American Journal of Software Engineering and Applications
2014; 3(3): 21-28
Published online July 10, 2014 (http://www.sciencepublishinggroup.com/j/ajsea)
doi: 10.11648/j.ajsea.20140303.11
ISSN: 2327-2473 (Print); ISSN: 2327-249X (Online)

Software reuse facilitated by the underlying requirement specification document: A knowledge-based approach

Oladejo F. Bolanle[1, 2, *], Ayetuoma O. Isaac[1, 2]

[1]Department of Computer Science, University of Ibadan, Ibadan, Nigeria
[2]University of Ibadan, UI, Ibadan, Nigeria

Email address:

fb.oladejo@ui.edu.ng (O. F. Bolanle), ayetuomaisaac@yahoo.com (A. O. Isaac)

To cite this article:

Oladejo F. Bolanle, Ayetuoma O. Isaac. Software Reuse Facilitated by the Underlying Requirement Specification Document: A Knowledge-Based Approach. *American Journal of Software Engineering and Applications.* Vol. 3, No. 3, 2014, pp. 21-28.
doi: 10.11648/j.ajsea.20140303.11

Abstract: Reinventing the wheel may not be appropriate in all instances of software development, and so, rather than do this, reuse of software artifacts should be embraced. Reuse offers certain benefits which include reduction in the overall development costs, increased reliability, standards compliance, accelerated development and reduced process risk. However, reusable software artifacts may not be considered useful if they cannot be accessed and understood. In this work, a knowledge based system was designed to capture requirements specification documents as abstract artifacts to be reused. Both explicit and tacit knowledge identification and acquisition- an important step in knowledge base development, was carried out through extraction from customer requirement documents, interviews with domain experts and personal observations. Protege4.1 was used as a tool for developing the Ontology. Web Ontology Language (OWL) was the search mechanism used to search the classified ontology to deduce reusable requirement components based on the underlying production rules for querying and retrieval of artifacts. Knowledge was formalized and result testing was carried out using software requirement specification documents from different domains. Result shows that only requirements with similar object properties called system purpose could really reuse such artifacts. The possibility of accessing more reusable artifacts lies in the update of the repository with more requirement specification documents. Scopes and purposes of previously developed software that would suit a proposed system in the same (or similar) domain would be found and consequently support the reuse of any of the end-products of such previously developed software.

Keywords: Knowledge Based System, Ontology, Reuse, Software, SRSR-Software Requirement Specification Reuse

1. Introduction

The design process in most engineering disciplines is based on reuse of existing systems or components. Mechanical or electrical engineers do not normally specify a design where every component has to be manufactured specially. They base their design on components that have been tried and tested in other systems. These are not just small components such as flanges and valves but include major subsystems such as engines, condensers or turbines. Software reuse “refers to the use of previously developed software resources in new applications by various users such as programmers and systems analysts” [1]. Considering the high cost and much stress involved in producing quality software one would expect that reuse should be a welcome idea to all stakeholders involved in the process, but research has shown the contrary; reuse has not been broadly applied across all spectrum of the industry.

Reuse-based software engineering is a comparable software engineering strategy where the development process is geared to reusing existing software. The paradigm shift to reuse-based approach in software development is as a result of the demands for reduction in the development and maintenance costs of software, faster delivery and improvement of the quality of software. More and more companies see their software as a valuable asset and are promoting reuse to increase their return on software investments. The tasks of maintaining the large collection of components and allowing the users to easily find out the components they need are critical to reducing the cost of

reuse. There is need for effective tools to support cataloguing the components and searching them. To achieve this, we might have to borrow the ideas and techniques from the field of Artificial intelligence (knowledge representation techniques), database management system field, and system science (techniques of building systems with components) [2].

2. Theoretical Background

2.1. Software Development Process and its Phases

Indeed, building computer software is an iterative learning process, and the outcome, something that Baetjer would call “software capital,” is an embodiment of knowledge collected, distilled, and organized as the process is conducted [3]. Software development process is a roadmap that provides the framework from which a comprehensive plan for software development can be established. The generic activities carried out in software development process include Software specification- this is where customers and engineers define the software to be produced and the constraints on its operations, software development- software is designed and programmed, software validation- software is checked to ensure it is what the customer actually asked for, software evolution- software is modified to suit changing customer’s needs and market requirement. A process model for software engineering- software engineering paradigm is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. A number of different process models for software engineering such as waterfall approach, component-based development, concurrent development model, the RAD model, the Prototyping model, Evolutionary software development process models-incremental model and spiral model, have been proposed, each exhibiting strengths and weaknesses[4], but all having a series of generic phases in common, prominent among these phases is a communication link between developers and customers for the purpose of requirement engineering covering feasibility study, requirement elicitation and analysis, requirement specification, validation and management, which goes to show that for a failure-free software product to be produced, a keen attention must be paid to this phase. Understanding what to build is one of the most tedious aspects of software development because sometimes customers do not really know what they want, so capitalizing on previously used abstract artifacts like requirement specification document may open the mind of software customers to more functionalities that could have been overlooked.

In a survey carried out by Standish group in 1994[5], [6] with over 350 companies, asking about the state of their over 8000 software projects, respondents were asked to explain the causes of failed software project. The top factors were reported as: Incomplete requirements-13.1%, Lack of user involvement-12.4%, Lack of resources-10.6%, Unrealistic expectations-9.9%, Lack of executive support-9.3%, Changing requirements and specifications-8.7%, Lack of planning-8.1%, System no longer needed-7.5%. Notice that some part of the requirements elicitation, definition, and management process is involved in almost all of these causes. Lack of care in understanding, documenting, and managing requirements can lead to myriad of problems: building a system that solves the wrong problem, that does not function as expected, or that is difficult for the users to understand and use [7]. This work focused on the reuse of requirement specification documents that have been used to implement successfully developed and operational software, enabling a developer to choose requirement specification that meet user needs, display well outlined components of requirement specification for ease of understanding and access in order to facilitate timely software development.

2.2. Software Reuse in Software Development Projects

Reuse is the default problem-solving strategy in most human activities, and software development is no exception. Software reuse means reusing the inputs, the processes, and the outputs of previous software development efforts; it is a means toward an end: improving software development productivity and software product quality. Reuse is based on the premise that deducing a solution from the statement of a problem involves more effort (labour, computation etc.) than inducing a solution from that to a similar problem, one for which such efforts have already been expended. While the inherent complexities in software development make it a good candidate for explorations in reuse, it is far from obvious that actual gains will occur. The challenges are structural, organizational, managerial, and technical [8].

With the increasing complexity in software systems, stakeholders in the business of software development need to get acquainted with vast amount of information and knowledge in various areas, likewise the need to store this knowledge for easy access for reuse. The knowledge gathered during the development stage can be a valuable asset for a developer as well as the software company. During the software development process, the management and maintenance of knowledge creation is a necessary thing. Then only that knowledge is integrated to develop the innovative concept from the older one. So the company must store and manage it for reuse [9].

Reusable artifacts can be software components, software requirement analysis manuals, and design models, database schema, objects, code documentation, domain architecture, test scenarios, and plans. The existing software can be from within a software system or other similar software systems or widely in different systems. For example, MS Office 2003 a tool to create and to edit different types of documents, worksheets, presentation slides and databases. They came up with the MS Office 2007 which is the latest version of it (as at 2007). Just like this there are so many examples we can consider as a Software Reuse [10].

Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes. For instance, Application system reuse- the whole of an application system may be reused by incorporating it without change into other systems, Component reuse- components of an application may be reused, Object and function reuse- components that implement a single function may be reused. A complementary form of reuse is concept reuse where, rather than reuse a component, the reused entity is more abstract and is designed to be configured and adapted for a range of situations. Concept reuse can be embodied in approaches such as design patterns, configurable system products and program generators. The reuse process, when concepts are reused, includes an instantiation activity where the abstract concepts are configured for a specific situation. Many techniques have been developed to support software reuse and these techniques exploit the facts that systems in similar application domain are identical and therefore have potential for reuse which is possible at different levels- from simple functions to complete applications [11].

According to Sommerville, (Sommerville, 2008), the followings are the number of ways to support software reuse (i.e. techniques for reuse):

i. Application product lines
ii. Aspect-oriented software development
iii. Configurable vertical applications
iv. Component-based development
v. Component frameworks
vi. COTS integration
vii. Design patterns
viii. Legacy system wrapping
ix. Program generators
x. Program libraries
xi. Service-oriented systems

Seeing that a huge number of techniques for reuse exist, it is therefore pertinent to figure out which is the most appropriate to use for a particular instance of reuse. When planning reuse, key factors one should consider are:

i. The development schedule for the software
ii. The expected software lifetime
iii. The criticality of the software and its non-functional requirements
iv. The background, skills and experience of the development team
v. The application domain and
vi. The platform on which the system will run.

Proponents claim that objects and software components offer a more advanced form of reusability, although it has been tough to objectively measure and define levels of reusability. Reusability implies some explicit management of build, packaging, distribution, installation, configuration, deployment, maintenance, and upgrade issues. If these issues are not considered, software may appear to be reusable from design point of view, but will not be reused in practice.

2.3. Knowledge Engineering Techniques

Knowledge refers to the perception an individual has about a fact or event in certain context [12]. For instance, a medical Doctor practices with the skill he possesses to treat and administer drugs to patients. Such knowledge is known as ‘know-how’. Also a procedure manual or recipe for a meal is another instance of knowledge which is regarded as ’know’. There are two main types of knowledge, explicit (objective) and tacit (subjective) knowledge. The various kinds of knowledge are illustrated in figure 1 [13]. Tacit knowledge refers to ‘know-how’ of an individual while explicit knowledge is the articulated knowledge in form of documents, operation manual, video, etc. Explicit knowledge could be readily transmitted across individuals formally and systematically.

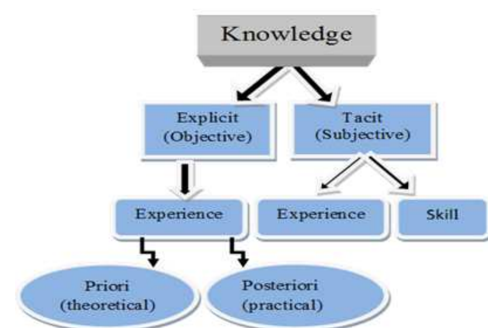

Figure 1. *Classification of knowledge [13].*

Knowledge Engineering is a branch of AI which analyzes the knowledge of a given domain and transforms it to a computable form for specific purpose. It entails knowledge representation which involves the translation of an informal specification to a formal (computable) one by a knowledge engineer who uses his wealth of background knowledge from reference sources or a domain expert [14]. There are certain guidelines for adequate knowledge representation.

2.3.1. Principles of Knowledge Representation

Certain factors that contribute to adequate representation of knowledge according to Randall in [14], are as follows.

Knowledge representation should serve as a surrogate (substitute/stand-in) for physical objects or events and the relationships amongst them with the aid of symbols and its links to model an external system.

It is a set of ontological commitments that determine various categories of objects of a domain.

It should describe the behavior and interaction amongst domain objects in order to reason about them.

Next is the fact that knowledge representation should be a medium for efficient computation which enables the encoding of represented knowledge in order to facilitate efficient processing with the aid of appropriate computing equipment.

Finally, it should be a medium of human expression in such a way to facilitate the understanding and

communication of both knowledge engineer and domain experts.

Knowledge representation is often augmented with reasoning (the process of applying knowledge to arrive at the conclusion) techniques: that is, provision of methods to handle the tracking of transition among system's properties or knowledge and underlying reasons for such transitions. There are two approaches to reasoning techniques, namely, declarative and procedural. The latter is similar to step-wise programming or algorithmic approach while the former requires the use of axioms or logical statements to describe specifications and theorem-proving technique to reason about knowledge [14]. The choice of appropriate techniques for representing and reasoning about domain knowledge actually depends on the nature of requirements for a knowledge-based system.

2.3.2. Development of Knowledge Based Systems

A Knowledge Based System (KBS) is a software application with an explicit, declarative description of knowledge for a certain application [15] (Speel et al, 2001) in Avram. There is no clear separation criterion between a KBS and an information/software system as almost all contain nowadays knowledge elements in them [20] (Schreiber et al, 1999) in [21] Avram. Conventional software applications perform tasks using conventional decision-making logic -- containing little knowledge other than the basic algorithm for solving that specific problem and the necessary boundary conditions. This program knowledge is often embedded as part of the programming code, so that as the knowledge changes, the program has to be changed and then rebuilt. Knowledge-based systems collect the small fragments of human know-how into a knowledge-base which is used to reason through a problem, using the knowledge that is appropriate.

The development process of a KBS is similar to the development of any other software system; phases such as requirements elicitation, system analysis, system design, system development and implementation are common activities. The stages in KBS development are: business modelling, conceptual modelling, knowledge acquisition, knowledge system design and KBS implementation [15] (Speel et al, 2001) in Avram.

A KBS is nowadays developed using knowledge engineering techniques (Studer et al 1998). These are similar to software engineering techniques, but the emphasis is on knowledge rather than on data or information processing. The central theme in knowledge engineering techniques is the conceptual modelling of the system in the analysis and design stages of the development process. Many of the knowledge engineering methodologies developed emphasizes the use of models (Common KADS, MIKE, and Protégé).

In the early stages, knowledge-based systems were built using the knowledge of one or more experts – essentially, a process of knowledge transfer (Studer et al, 1998). Nowadays, a KBS involves "methods and techniques for knowledge acquisition, modelling, representation and use of knowledge" (Schreiber et al, 1999) in Avram

In current practice the transfer of expertise from a domain specialist to a knowledge-based system involves a computer scientist intermediary–or knowledge engineer. The specialist and the engineer discuss the domain in a series of interactions. During each interaction, the engineer gathers some understanding of a portion of the specialist's knowledge, encodes it in the evolving system, discusses the encoding and the results of its application with the specialist, and refines the encoded knowledge. The process is a painstaking one–expensive and tedious. As a result, one of the foremost problems that have been identified for KBSs is the knowledge acquisition bottleneck (Reid, 1985). The shift towards the modelling approach has also enabled knowledge to be re-used in different areas of one domain (Studer et al, 1998). Ontologies and Problem-Solving Methods enable the construction of KBSs from components reusable across domains and tasks.

2.4. Review of Related Works

According to [16] in a paper titled *A Pragmatic Approach to Software Reuse,* published in a Journal of Theoretical and Applied Information Technology, the reliability level of every reusable software artifact (requirement specification document, in this instance), is enhanced by a successful reuse and this success in turn increases the usefulness of such artifact in the reuse repository (such as a knowledge based system), and ultimately, the risk of failure of the resulting software product developed from such artifact is reduced. With the availability of a knowledge based system that serves as a repository for software requirement specification documents, which is a basis for building software adequately reflects the user's need and developer's technical know-how, timeliness in development and cost reduction would be facilitated while validation and verification of software will be enhanced likewise. Higher scheduling accuracy of the various tasks in the software development process is possible due to the reuse of process materials along with a better understanding of the product domain; therefore categorizing requirement specification document in the knowledge base along domain line is worthwhile.

Since the process has been completed before, project managers, having access to previous projects' scheduled and actual hours for production can adjust their current schedule based on previous performance and the amount of reusable artifact in the repository they intend to use. According to Jalender et al., the most substantial but not immediate benefits of reuse is derived from product line approach where a common set of reusable software assets act as a base for subsequent similar software product in a given functional domain. It was posited that the upfront investments required for software reuse are considerable and need to be duly considered prior to attempting a software reuse initiative, repositories of software assets

must be created and maintained.

Lethal and Carl (1997) wrote on Automatically Identifying Reusable OO Legacy Code where they are of the view that much object-oriented code has been written without reuse in mind, making identification of useful components difficult. The Patricia system (a tool for object oriented program understanding) automatically identifies these components through understanding comments and identifiers. According to [17], aspects of Object oriented code such as classes, inheritance, and parametric polymorphism underline the need for good, semantically based tools to aid in the understanding, and thus the reuse, of Object oriented code. The paper stated that to determine whether a code component can be reused in a particular domain, or area of application, these semantically based tools must answer two questions: Are the purpose and capabilities of the code component useful in the current domain? Is the quality of the code component sufficient for the needs of the current domain?

Completely understanding what capabilities a class provides involves gathering information from a variety of sources, including the source code, user documentation (such as manuals), and documentation for requirements and design specifications.

In a research work "Towards Principles for the Design of Ontologies Used for Knowledge Sharing", [18] analyzed design requirements for shared ontologies and a proposal for design criteria to guide the development of ontologies for knowledge-sharing purposes. A usage model for ontologies in knowledge sharing was described and some design criteria based on the requirements of this usage model were proposed.

In a paper "Using Ontologies for Knowledge Management: An Information Systems Perspective", [19] surveyed some of the basic concepts that have been used in computer science for the representation of knowledge and summarized some of their advantages and drawbacks. The survey classifies the concepts used for knowledge representation into four broad ontological categories viz: Static ontology which describes static aspects of the world, i.e., what things exist, their attributes and relationships, A Dynamic ontology, which describes the changing aspects of the world in terms of states, state transitions and processes, Intentional ontology, which encompasses the world of things agents believe in, want, prove or disprove, and argue about, and Social ontology which covers social settings, agents, positions, roles, authority, permanent organizational structures or shifting networks of alliances and interdependencies. They advocated a complementary use of concepts and techniques from information science and information systems in knowledge management as a result of the vast, complex and dynamic information environments. The ontology approach from information modeling described in this paper derives its strength from the formalization of some domain of knowledge; however, many domains resist precise formalization. In each domain, there are points at which formalization becomes more of a straightjacket than a liberating force. The challenge therefore is not so much to decide which approach is better, but to develop techniques for the various approaches to work closely together in a seamless way. It was posited that the key to providing useful support for knowledge management lies in how meaning is embedded in information models as defined in ontologies.

3. Research Methodology

3.1. Conceptual Framework for Knowledge Engineering for Reuse

In an attempt to model the knowledge base for the reuse of software requirement specification, the conceptual framework for knowledge Engineering for reuse is first established. This is represented in figure 2, describing how knowledge is represented and computed to produce output for utilization as a solution to an existing problem.

How do we represent what we know? Knowledge is a general term. An answer to the question, "how to represent knowledge", requires an analysis to distinguish between knowledge "how" and knowledge "that". Knowing how to do something, for example, "how to operate a machine" is a Procedural knowledge. Knowing that something is true or false, for instance, "the temperature limit for a machine in operation" is a Declarative Knowledge.

Knowledge and Representation are two distinct entities. They play a central but distinguishable role in intelligent systems. Knowledge is a description of the world. It determines a system's competence by what it knows.

Representation is the way knowledge is encoded. It defines a system's performance in doing something. A good representation enables fast and accurate access to knowledge and understanding of the content. Knowledge representation can be considered at two levels:

a. Knowledge level at which facts are described, and

b. Symbol level at which the representations of the objects, defined in terms of symbols, can be manipulated in the programs.

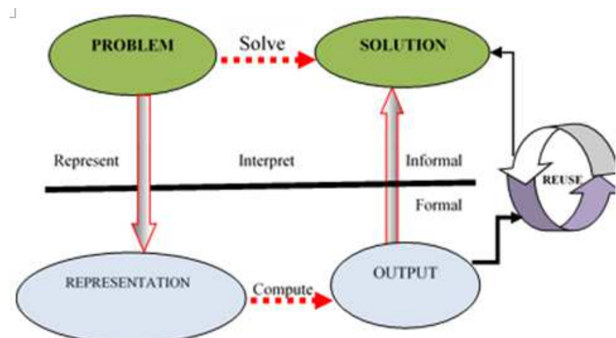

Figure 2. *Knowledge Representation Framework [22] Extended for Reuse*

Different types of knowledge require different kinds of representation and reasoning. The knowledge representation models/mechanisms are often based on: (i) Logic, (ii) Rules, (iii) Frames and (iv) Semantic Network.

Fig. 2 shows a description of framework adapted from [22] for knowledge representation and reuse of such knowledge.

The computer requires a well-defined problem description to process and provide well-defined acceptable solution from a reused component. To collect fragments of knowledge we need first to formulate a description in our spoken language and then represent it in formal language so that computer can understand. This is where ontology comes in into modelling the contents of the knowledge base. The computer can then use an algorithm to compute an answer as illustrated in fig. 2. The steps are:

The informal formalism of the problem takes place first. It is then represented formally in ontology and the knowledge base produces an output upon query. This output can then be represented in an informally described solution that user (software engineers) understands or checks for consistency in line with initial customer's requirements.

It is noteworthy however to state that problem solving requires formal knowledge representation, and conversion of informal knowledge to formal knowledge, as well as conversion of implicit knowledge to explicit knowledge.

3.2. Formalization of Knowledge Engineering Framework for Software Reuse

The Finite Automaton (FA) in fig. 3 is a five tuple (Q, ∑, δ, So, F) describing the process of transmission of software requirement data from one state to another, undergoing refinement and necessary adjustment as required where,

Q is the set of all states in the automaton represented by circles in fig. 3, thus:

$Q = \{S_0, S_1, S_2, S_3, S_4\}$ where:

S_0= Initial customer specification information

S_1= Designer's specification

S_2= Agreed specification (Software Requirement Specification)

S_3 = Formalized knowledge in the knowledge based system

S_4 = Reusable knowledge (for future specifications)

∑ is the string of valid characters that can occur in the input stream. Typically, ∑ is the union of the edge labels in fig. 3.

$\delta: Q \times \Sigma \rightarrow Q$ is the transition function for the automaton. It depicts the state changes induced by an input character string for each state; δ is represented by the labeled edges that connect states in fig. 3.

$\{(\langle S_0, T_1\rangle \rightarrow S_1),$

$(\langle S_1, T_2\rangle \rightarrow S_0),$

$(\langle S_1, T_3\rangle \rightarrow S_2),$

$(\langle S_2, T_4\rangle \rightarrow S_1),$

$(\langle S_2, T_5\rangle \rightarrow S_3),$

$(\langle S_3, T_6\rangle \rightarrow S_2),$

$(\langle S_3, T_7\rangle \rightarrow S_4),$

$(\langle S_4, T_8\rangle \rightarrow S_2)\}$

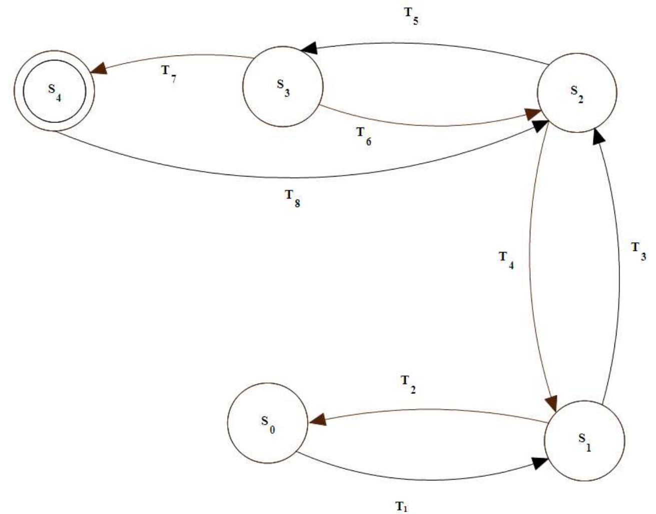

Figure 3. The Finite Automata showing framework for Reuse of Software artifact (requirement engineering

3.3. Knowledge Base Development for Software Requirement Specification for Reuse (SRSR)

The approach to reuse adopted is knowledge-based; a software system with an explicit, declarative description of knowledge for diverse domains. We would expect to find artifacts from which self-contained applications can be constructed based on certain characteristics and the goal (purpose for utilization, driven by requirement specification). The knowledge based system was implemented using protégé 4.1.

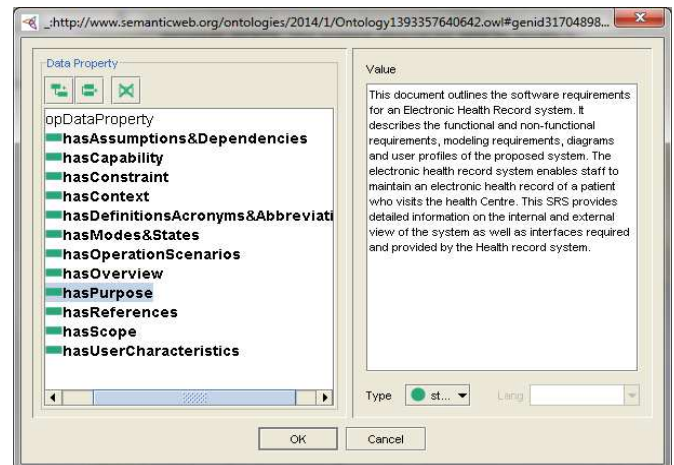

Figure 4. *An Existing Health Records System SRS showing its 'PURPOSE'*

4. Research Findings and Result

The ontological tool used for the ontological architecture of the knowledge based system is Protégé 4.1. This produces a generalized class of ontology that describes various requirement specification documents for different domains of software. The object properties of the various classes are inherited from the super class called Software Requirement Specification and these properties reflect the content specified by IEEE standard 1233[23]. Fig. 4 shows the screen shot of a requirement specification document retrieved to view its purpose, which reflects the functional and non-functional requirements of the software being developed.

Artifacts stored in this repository possess the attributes shown in table 1, and these make them suitable for reuse.

Table 1. *Features of suitability of the artifacts for reuse*

Attributes	Comments
Simplicity	Minimum and explicit artifact interfaces which encourage developers to use artifacts, simple and easy to understand artifacts can also be easily modified by developers to suit new applications.
Expressiveness	They are of general utility and of adequate level of abstraction, so they could be used in many different contexts within their domains.
Definite	They are constructed and documented with clarity of purpose, their capabilities and limitations are easily identifiable, interfaces, required resources, external dependencies and operational environments are specified, and all other requirements are explicit and well defined.
Additive	It is possible to seamlessly compose these existing artifacts into new products or other reusable components, without the need for massive software modifications or causing adverse side effects
Easily Changeable	Certain type of problems will require artifacts to be adapted to the new specifications, such changes should be localized to the artifact and require minimum of side effects
Unambiguous	Each requirement is stated in such a way so that it can be interpreted in only one way without ambiguities.
Organized	The requirements and sub-components are well structured in the ontology

5. Conclusion

Software reuse, as appealing as it appears, if it is not carefully implemented, the cost involved in software development using reuse may be much more than that incurred when software is developed from scratch. Therefore, reuse that is facilitated by the knowledge based repository of the underlying the requirement specification documents of previously developed software product is like building a strong foundation for a complex structure, and this gives reliability and assurance of quality. The knowledge based system uses semantic approach to search for reusable requirement components whose product can be partly or entirely utilized for developing a proposed system. With software domain specific repository available, based on the reference architecture and on the requirements we would be able to locate and reuse some domain specific reusable components. This work can be extended to see how the reuse of requirement specification documents alters the conventional software development life cycle.

References

[1] Y. Kim and E.A. Stohr, “Software Reuse: Survey and Research Directions,” Journal of Management Information Systems, 1997.

[2] R. B. Victor, B. John, G. J. Bok, and H.R. Dieter, “Software Reuse: A framework for Research,” Department of Computer Science, University of Maryland, 2002.

[3] H. Baetjer, Jr., “Software as Capital,” IEEE Computer Society Press, 1998, p. 85.

[4] R. S. Pressman, “Software Engineering. A Practitioner’s Approach,” 5th ed., McGraw-Hill series in Computer science, 2001.

[5] Standish Group. The CHAOS Report. Dennis, M.A: The Standish Group, 1994.

[6] Standish Group. “The Scope of Software Development Project Failures,” Dennis, MA: The Standish Group, 1995.

[7] S.L. Pfleeger and M.A. Joanne, “Software Engineering: Theory and Practice.” 4th ed., Pearson Higher Education, 2010.

[8] M. Hafedh, M. Fatma, and M. Ali, “Reusing Software: Issues and Research Directions,” IEEE Transactions on Software Engineering, Vol. 21, No. 6, June 1995.

[9] G.N.K. Suresh, and S.K. Srivatsa, “Analysis and Measures of Software Reusability.” International Journal of Reviews in Computing, 2009.

[10] P. Nandish, “Software Reuse and/or Software Complexity Management.” Networks on Chips, 2008.

[11] I. Sommerville, “Software Engineering,” 8th Edition, (Addison-Wesley Publishers Ltd.,), 2008, pp. 415 - 438.

[12] B.F Oladejo & A.O. Osofisan, “A Conceptual Framework for Knowledge Integration in the Context of Decision Making Progress,” African Journal of Computer & ICT, vol. 4, No. 2. Issue2. Pp.25-32, 2011.

[13] B. F. Oladejo, V. T. Odumuyiwa, and A. A. David, “Dynamic Capitalization and Visualization Strategy in Collaborative Knowledge Management system for EI process,” World Academy of Science, Engineering and Technology pp66, 2010.

[14] J. F. Sowa, “Knowledge Representation: logical, philosophical, and computational foundations,” Brooks Cole Publishing Co., Pacific Grove, CA. 2000.

[15] P. Speel, A. Th. Schreiber, W.V. Joolingen, and G. Beijer, "Conceptual Models for Knowledge-Based Systems." Encyclopedia of Computer Science and Technology, Marcel Dekker Inc., New York, 2001

[16] B. Jalender, A. Govardhan, and P. Premchand, "A Pragmatic Approach to Software Reuse." *A Journal of Theoretical and Applied Information Technology*, 2010.

[17] H.E. Lethal, and G. D. Carl, "Automatically Identifying Reusable OO Legacy Code." *University of Alabama, Huntsville*, 1997.

[18] T.R. Gruber, "Toward Principles for the Design of Ontologies Used for Knowledge Sharing." 1993.

[19] J. Igor, M. John, and Y. Eric, "Using Ontologies for Knowledge Management: An Information Systems Perspective. University of Toronto, Toronto, Ontario, Canada. 1999.

[20] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W.V. de Velde, and B. Wielinga, "Knowledge Engineering and Management: The Common KADS Methodology," Massachusetts: MIT Press, 1999.

[21] G. Avram. Empirical Study on Knowledge Based Systems. The Electronic Journal of Information Systems Evaluation, Vol. 8, Iss.1, pp 11-20, available online at www.ejise.com, 2005

[22] D. Poole, "Knowledge Representation Framework," 1998.

[23] IEEE-Std 1233, IEEE Guide for Developing System Requirements Specifications, 1998.