International Journal of Information Technology and Knowledge Management
January-June 2008, Volume 1, No. 1, pp. 169-178

A REVIEW OF APPROACHES TO SOFTWARE REUSE

Rajender Nath & Harish Kumar

Abstract

Benefits provided by software reuse are well known to software developers. Through software reuse, software developers can produce high quality software in comparatively lesser time and lesser budget. The researchers have developed a number of software reuse techniques. This paper presents an overview of widely used software reuse approaches and gives a roadmap to future research in this field.

Keywords: Software reuse, reuse approaches, domain engineering, reuse architecture.

1. Introduction

The idea of software reuse seems simple, but its implementation is very difficult. Today, the majority of the corporations do not have formal reuse programs. The concept of software reuse came into existence about four decades ago when McIlroy gave an idea of applying reuse in software in a NATO software engineering conference in 1968. No significant work was carried out for about one decade. About thirty years before, some work on reuse-based software technology started [1]. Although reuse is four decades old, it is not so popular as of yet. Researchers cannot make practical use of reuse technology; they can only develop a new technology and can encourage potential users to adopt it.

2. Software Reuse Approaches

There are a number of approaches of reusing software reported by the researchers in the literature. We can broadly divide these reuse approaches into three broad categories: (i) component-based software reuse, (ii) domain engineering & software product lines and (iii) architecture based software reuse. If different software systems are divided into atomic components, it is found that different software systems have some common components. Component-base reuse is based on this very fact. In component-based reuse, a repository of these common atomic components is built and new software system is built by selecting appropriate component from repository besides developing new components every time. Domain engineering captures the commonalities and variability in a set of software systems and uses them to build reusable assets. Architecture-based reuse extends the definition of reusable assets to a whole design or subsystem composing of components and relationship among them. The three approaches are not mutually exclusive and in many cases a combination of these

approaches is used [2]. More work has been done on component-based software reuse. Significant work has been done on domain engineering and software product lines but architecture-based reuse still requires more attention of the researchers. In the following, these three approaches are presented in sufficient details.

2.1 Component-Based Software Reuse

Most of the engineering disciplines support reuse. For example, in mechanical engineering, some parts of an existing machine can be used to fabricate a new machine. Like it, some components of one software systems may be used to build other software system. In component-based reuse, a repository of small independent software components is built and a searching mechanism to match the requirement with stored components is also developed. Finding the best match is the first and the most important step in reuse [3]. Requirement is to be specified without ambiguity. Some formal specification language like Z or Unified Modeling Language may be used to specify the requirement. Stored components should also have some parameters that can be matched with the requirement and selected for reuse. For a given requirement, a number of components may be retrieved that fulfill the requirement more or less. Out of these retrieved components, one component that best matches is selected for reuse [4].

A software component is a prewritten element of software with clear functionality and a well-defined interface that identifies its behavior and interaction mechanism. McClure [5] identifies several properties a software component must have to be reusable.

These properties include a set of common functionality, a well-defined interface that hides implementation details, the ability to inter-operate with other components, and the ability to be reusable in several different software systems.

The biggest challenge in component based software reuse is managing repository of a large number of reusable components and developing an efficient retrieval mechanism. Several methods have been proposed in literature since the early days of component reuse. The most common approach is discussed below.

2.1.1 *Building a Repository and Developing Search Mechanism*

Structure of the repository is most important factor in obtaining good retrieval results. Even though some retrieval algorithms can provide adequate effectiveness with minimal indexing and structuring efforts. Poorly structured repository with poor indexing will not have a good retrieval performance regardless of the retrieval algorithms used [6].

Software repository indexing methods can be divided into two main categories: manual and automatic indexing [2]. Common examples of manual indexing include enumerated and faceted classification. In automatic indexing, most common method is free-text indexing.

Enumerated Classification involves defining a hierarchical structure of categories and subcategories where actual components are at the leaf level of the classification tree. The hierarchical structure is easy to use and understand and provides a natural searching method of navigation in the classification tree. However, the main problems with this classification include its inflexibility and difficulty to change as the indexing domain changes. It also requires extensive domain analysis before classifying components into exclusive categories. Furthermore, retrieving classified components is only easy for users who understand the structure and contents of the repository; it is not easy for the users who are unfamiliar with the structure of the repository. Single classification cannot represent complicated domains. Hence no one classification is correct under all circumstances.

Ruben Prieto-Diaz proposed faceted classification in 1987 [7]. It defines a set of mutually exclusive facets combined to completely describe all the components in a domain. Each facet can be described by a group of different terms. Users search for components by choosing a term to describe each of the facets. A faceted classification scheme gives freedom to create complex relationships by combining facets and terms. Modification in faceted classification is much easier than a hierarchical classification because one facet can be changed without affecting others in the classification scheme. But the users must be familiar with the structure of the facets and their terms. Also, sometimes it is not clear that what combination of terms to use in the search.

Free-text indexing uses the words in a document to create index terms. Index term lists contain all words and their number of occurrences in the document. Most free-text indexing techniques work with a stop list that prevents certain high frequency words such as "a", "the", and "is" from being indexed. To find a document that contains a set of keywords, users specify these keywords that are matched against the index term list to find the best matching documents. Even though easy to build and works very well in many applications including online search engines, free-text indexing is not suitable for indexing code and other software artifacts. Free-text indexing relies heavily on natural language rules and use statistical measures that need large bodies of text to be accurate. However code has arbitrary rules of grammar and allows the use of non-words and abbreviation and may contain minimum or no documentation [6].

2.1.2 *Issues in Repository Retrieval*

Effective storage and efficient retrieval of components are the most essential parts of any repository. One major issue in repository retrieval is reuse barrier. Most current reuse repository systems are designed as a separate process, independent of current development processes and tools assuming that software developers will start the reuse process when they need to. This assumption, however, is often not correct. Several empirical studies suggested that "no attempt to reuse" is the most common reason of failure in the reuse process. Because of the large and constantly changing component

repositories, programmers often fail to correctly anticipate the existence of reusable components. When developers do not believe that a component exists, they will not even make an attempt to find it [8]. Therefore, the reuse process will never be initiated in the first instance.

A second issue with repository retrieval systems is vocabularies and ill-defined queries submitted by non-expert users. This issue is a consequence of another faulty assumption in repository retrieval systems that developers know exactly what they need and can translate it into well-defined queries. In most of the cases, developers have a vague idea of what they need to find and cannot express it in a clear set of terms. Even if developers know what they are looking for, they may not have the knowledge needed to express it in terms and vocabularies, the retrieval system uses. The same developer might use different terms to describe the same component at different times. Furthermore, repositories are usually indexed by experts who might use terms that non-expert developers are not familiar with [6]. According to a study conducted in 2006, novice developers are more likely to make use of reuse processes than expert ones [9]. Therefore, novice developers should be given preferences in reuse programs.

2.1.3 *Available Retrieval Tools*

Several tools have been proposed in literature to facilitate software reuse process [2]. Expert system based reuse tools were developed in mid 1990s [10]. Many other tools were also proposed to address the retrieval issues described above. Two of these tools, CodeFinder and CodeBroker are discussed briefly.

CodeFinder was first proposed by Henninger in 1995 [6] to support the process of retrieving software components. It is useful when users are not familiar with terms used in the repository and queries submitted by users are ill defined. It employs two techniques: intelligent retrieval method that finds information associatively related to the query, and query construction supported through incremental refinement of queries. The main idea is to create a retrieval method that utilizes minimal, low cost repository structure and evolves to adapt to user needs.

The proposed system is composed of three main parts: a tool (PEEL) to initially populate the repository with reusable components, a searching mechanism, and an adapting tool to refine the repository when needed.

PEEL (Parse and Extract Emacs Lisp) is a semiautomatic tool that translates Emacs Lisp (a variant of Lisp) source code files into individual reusable components of functions and routines. These components are indexed in the repository using terms extracted from variable and function names and comments preceding definitions.

The system provides a user interface that implements the searching and browsing mechanism to allow the user to view and browse the hierarchy of the repository as well as submit search queries. Each time the user submits a query, the system responds with

the retrieved items and a list of terms used to index each item. This technique helps users that are not familiar with the terms of the repository system. As the user explores the information space, they become more familiar with repository structure and terms and incrementally refine their queries based on the previous results. In case, when queries are ill defined, the CodeFinder retrieval method does not try to exactly match the user queries with the item descriptions. It uses association by applying associative network, to retrieve items that are relevant to a query and may not exactly match it.

CodeFinder, however, does not address the reuse barrier issue discussed in section 2.1.2. The user interface and tools are separated from the Lisp development environment. It relies on the developer to initiate the reuse process by switching between the two environments (development and reuse) whenever needed. Many developers overestimate the cost of switching and searching for reusable item and do not attempt using reuse. In 2001, Yunwen proposed an *active* and *adaptive* reuse repository system called "CodeBroker" to address this issue [8].

CodeBroker repository system is not a standalone tool; it is integrated into the development process running continuously in the background of the development environment. It is an active system that presents information when identifies a reuse opportunity without waiting for users to submit explicit queries. It is also an adaptive system that captures developer's preferences and knowledge level and saves them to a profile. This profile is used to adapt the system's behavior for each user. User profiles can be explicitly modified by users (adaptable) or implicitly updated by the system (adaptive).

The system's repository is composed of Java API and General Library. Its architecture consists of three software agents: listener, fetcher and presenter. Listener is a background running process that captures developers' needs for reusable components and formulates reuse queries. Whenever a developer finishes a document comment or a function definition, listener automatically extracts the contents and creates a concept query and passes it to fetcher. Fetcher executes the query and retrieve relevant components from repository based on concept similarity to the comment text or constraint compatibility to the function signature. Presenter displays retrieved components to the user in the RCI-display taking into consideration user profile settings.

2.2 Domain Engineering and Software Product Lines

Domain engineering captures the commonalities and variability in a set of software systems and uses them to build reusable assets. Most organizations work only in a small number of domains. For each domain they build a family of systems that are based on specific customer needs. Domain engineering deals with identifying the common features of existing systems within a particular domain and using these features as a common base to build a new system in the same domain [2]. This will result in higher efficiency and productivity.

Domain engineering has two stages: domain analysis and domain implementation. Domain analysis is the process of examining the related systems in a domain to identify the commonalities and variability. Domain implementation is the employment of that information to develop reusable assets based on the domain commonalities and uses these assets to build new systems within that domain. Shiva et al. have mentioned some domain engineering approaches in [2]. A few of these are described below.

2.2.1 *DAREm*

Domain Analysis and Reuse Environment (DARE) is a tool developed in 1998 to support capturing domain information from experts, documents and code. Captured domain information is stored in a database also called domain book that typically contains a generic architecture for the domain and domain-specific reusable components. DARE also provides a library search facility with a window interface to retrieve the stored domain information [11].

2.2.2 *FAST*

Family-Oriented Abstraction, Specification and Translation (FAST) is a system family generating method based on an application modeling language (AML). It was developed by Weiss and Lai at AT&T and continued to evolve at Lucent Technologies [12].

FAST is a systematic way of guiding software developers to create the tools needed to generate members of a software product line using a two-phase software engineering method: domain engineering phase and application engineering phase. The domain-engineering phase defines the product line requirements and design through a Commonality and Variability Analysis and develops a small special purpose language to describe these commonalities and variability. The application-engineering phase uses the application modeling language as the basis to generate the requirements and design of new family members.

2.2.3 *FORM*

Kyo C. Kang and others in Pohang University of Science and Technology presented a Feature Oriented Reuse Method (FORM) as an extension to the Feature Oriented Domain Analysis (FODA) method as FORM incorporates a marketing perspective and supports architecture design and object oriented component development [13]. FORM is a systematic method of capturing and analyzing commonalities and differences of applications in a domain and using the results to develop domain architectures and components. It starts with feature modeling to discover, understand, and capture commonalities and variability of a product line.

2.2.4 *KobrA*

KobrA is a German acronym (Komponentenbasierte Anwendungsentwicklung) stands for component-based application development [r 12, 2]. The KobrA method offers a

full life-cycle approach that integrates the advantages of several advanced software engineering technologies including product line development, component based software development and frameworks to provide a systematic approach to developing high-quality, component-based application frameworks [14]. KobrA is based on the principle of strictly separating the product from the process. The products of a KobrA project are defined independently of, and prior to, the processes by which they are created, and effectively represent the goals of these processes. Furthermore, KobrA is "technology independent" in the sense that it can be used with all three major component implementation technologies CORBA, JavaBeans and COM.

2.2.5 *PLUS*

Product Line UML-Based Software Engineering (PLUS) is a model-driven evolutionary development approach for software product lines. Gomaa introduced it in 2004 as an extension to the single system UML-base modeling methods to address software product lines [15]. In addition to analyzing and modeling a single system, PULS also provides a set of concepts and techniques to model the commonality and variability explicitly in a software product line. With these concepts and techniques, object oriented requirements, analysis and design models of software product lines are developed using UML 2.0 notation.

2.3 Architecture-Based Software Reuse

Effective reuse depends not only on finding and reusing components, but also on the ways those components are combined [16]. Architecture of software system is composed of its software components, their external properties, and their relationships with one another. Architecture-based reuse extends the definition of reusable assets to include these properties and relationships [2]. Since the late 1980's software architecture has been recognized as an important consideration for reusing software. Architectural decisions as they occur early in the software lifecycle, have a strong impact on system quality attributes. Architectural decisions are also difficult to change late in the lifecycle.

Shaw classified software architecture into common architecture styles where every style has four major elements: components, connectors, a control structure and a system model. Connectors mediate interactions among components. Control structure governs execution and rules about other properties of the system and system model captures the intuition about how the previous elements were integrated. Pipeline, data abstraction, implicit invocation, repository, and layered architecture are some of the popular architecture styles described by Shaw in [16]. Software architecture may be explored at different levels of abstraction. Architecture styles are commonly used in software and examine quality attributes related to each style. At a lower level of abstraction than style, architectural patterns that commonly occur in various design problem domains such as client-server architectures, proxies, etc. may be identified. In theory, these architecture patterns can be defined by applying a combination of architecture styles.

Applying a combination of architecture styles create architectural patterns [17]. An architectural pattern is a high-level structure for software systems that contains a set of predefined sub-systems, defines the responsibilities of each sub-system and details the relationships between sub-systems. Layers, Pipes and Filters, and Blackboard are some of the common patterns described by Buscmann *et al.*, in [18].

Using architecture patterns, reference architectures for an application domain or a product line can be built. These architectures embody application domain-specific semantics and quality attributes inherited from the architecture patterns. Application architectures may be created using domain architectures. Examples of domain architectures are reported in [19].

Platform architectures are middleware on/with which applications and components for implementation of an application can be developed. Examples of these are CORBA, COM+, and J2EE. A platform architecture selected for implementation of applications in a domain may influence architectural decisions for domain architecture. For example, transaction management is supported by most of platform architectures and domain architecture may use facilities provided by the platform architecture selected for the domain.

The relationships between these concepts related to architectures are summarized in following figure [17].

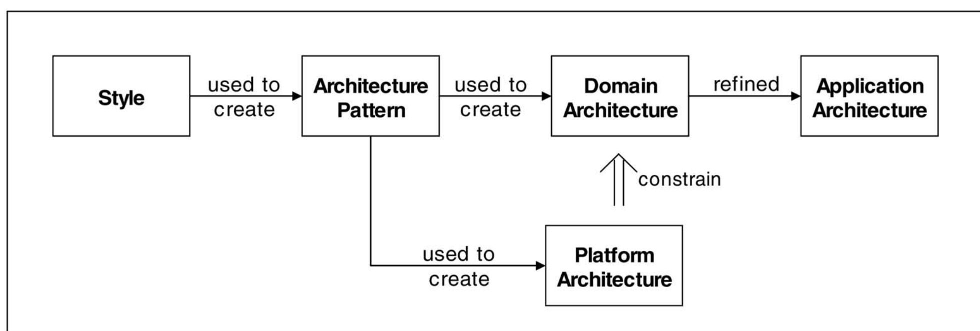

Figure 1: Architecture Concepts

Software systems in the same domain have similar architectural patterns that can be described with a generic architecture i.e. domain architecture. Domain architecture is then refined to fit individual application in the domain creating application architecture [12].

Software architecture may be based on services. This leads to a new approach known as Service-Oriented Architecture (SOA). SOA brought new chances to improve the development of reusable components. A service as a reusable unit is more appropriate for reusing software, because the logic of a service is generally so complicated that it is much easier for the service requesters to understand the contract than to implement the

service by themselves. Building reusable components with service-oriented architectures has been elaborated by Haibin Zhu in [20].

Software architecture is not only concerned with structure and behaviour, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns. Unified Modeling Language (UML) is best-suited modeling language to help building accurate and complete architecture of a software system. Use cases in UML can be used to build unambiguous and easily understandable architecture of a software system. A use case is a description of set of sequence of actions that a system performs. Framing reuse oriented software architecture using use cases has been described by one of the inventor of UML, Ivar Jacobson in [21].

Conclusion

Most of the corporate know the benefits of reuse but they hesitate to adopt it as it happens in the case of every new technology. The position of most software developers and their management on reuse is that they do not know how to implement it. That is, theoretically sufficient work has been done on software reuse but practically; it is still in an infancy stage. One of the major problems in software reuse implementation is that the most of the software methodologies do not include software reuse in the development process. The methodologies have not explicitly defined the where, when, and how to practice reuse as part of the development process. Furthermore, the risk and initial cost involved in implementing reuse is a barrier in not promoting software reuse. Actual benefits of reuse starts after a long period; consequently use of software reuse is ignored by most of the organizations.

References

[1] Yong-liu, & Aiguang-yang, (2007), Research and Application of Software-reuse, Eighth ACIS *International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/ Distributed Computing, IEEE*, pp. 588-593.

[2] Sajjan G. Shiva, & Lubna Abou Shala, (2007), Software Reuse: Research and Practice, *International Conference on Information and Technology, (ITNG'07), IEEE*.

[3] William N. Robinson, & Han G. Woo, (2004), Finding Reusable UML Sequence Diagrams Automatically, *IEEE Software*.

[4] Rajender Nath, & Harish Kumar, Issues in Software Reuse, National Conference on Futuristic Trends in Engineering and Technology-IFTE'07, JCD Vidyapeeth, Sirsa (Haryana).

[5] C. McClure., (2001), Software Reuse: A Standards-based Guide, Wiley-IEEE Computer Society Pr, New York.

[6] Henninger S., (1997), An Evolutionary Approach to Constructing Effective Software Reuse Repositories, *ACM Transaction on Software Engineering and Methodology*, **6**(2), 111-140.

[7] R. Prieto-Daz., (1987), Classifying Software for Reusability, *IEEE Software*, **4**(1), 6-16.

[8] Ye Yunwen., (Jan. 3-6, 2001), An Active and Adaptive Reuse Repository System, Proceedings of 34th Hawaii *International Conference on System Sciences (HICSS-34)*, 9065-9075.

[9] Desouza, Kevin, Awazu, Yukika, & Tiwana, Amrit, (2006), Four Dynamics for Bringing Use Back into Software Reuse, *Communications of the ACM*, **49**(1), 97-100.

[10] P. Wang, & S. G. Shiva, (March 1994), A Knowledge-based Software Reuse Environment, *IEEE Southeastern Symposium on System Theory*, 276-280.

[11] W. Frakes, R. Prieto-Diaz, & C. Fox, (Nov. 1997), DARE-COTS. A Domain Analysis Support Tool, Computer Science Society, 1997. Proceedings, *XVII International Conference of the Chilean*, **10-15**, 73-77.

[12] J. Coplien, D. Hoffman, & D. Weiss, (Nov.-Dec. 1998), Commonality and Variability in Software, *IEEE Software*, **15**(6), 37-45.

[13] K. C. Kang, J. Lee, & P. Donohoe, (July-Aug. 2002), Featured Oriented Product Line Engineering, *IEEE Software*, **15**(6), 58-65.

[14] C. Atkinson, J. Bayer, & D. Muthig, (2000), Component-based Product Line Development: The KobrA Approach, *1st International Software Product Line Conference*, Denver, 289-310.

[15] H. Gomaa, (2004), Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures. Addison-Wesley.

[16] M. Shaw, (April 1995), Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging, Proceeding of *IEEE Symposium on Software Reusability*, 3-6.

[17] W. Frakes, & Kyo Kang, (2005), Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, **31**(7), 529-536.

[18] F. Buschmann *et al*., (1996), Pattern-Oriented Software Architecture, Chichester, UK; New York: Wiley.

[19] W. Tracz, (July 1995), "DSSA (Domain-specific Software Architecture) Pedagogical Example," *ACM SIGSOFT Software Eng*. Notes, **20**(3), 49-62.

[20] Haibin Zhu, (2005), Building Reusable Components with Service-Oriented Architectures, 0-7803-9093-8/05, *IEEE*, 96-101.

[21] Ivor Jacobson, (February 2008), Use Cases and Architecture, CSI Communication, **31**, 6-11.

Rajender Nath
Reader, Department of Computer Science and Applications,K. U., Kurukshetra
E-mail: r-nath_2k3@rediffmail.com

Harish Kumar
Lecturer, Department of Computer Science and Engg, CDLU, SIRSA
E-mail: rohilharry@rediffmail.com