Volume 4, Issue 10, October 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

A Framework for Software Reuse and Research Challenges

Sathish Kumar Soora

Abstract: Software reuse is the use of existing software or software components to build new software and reuse ideas with the ability to combine independent software components to form a larger unit of software. The key idea in software reuse is domain engineering. Most software systems are not new but they are the variations of the already built software systems. Software reuse improves the quality and productivity of the software production process. This paper briefly summarizes the current research status in the field of software reuse and major research contributions. Some future directions for research in software reuse are also discussed.

Index Terms— Software reuse, Domain engineering, Software components.

I. INTRODUCTION

Software reuse is the process of creating software systems from existing software rather than building them from scratch. The software reuse recognized as having significant potential to improving software development productivity and software quality. At a high-level, software reuse consists of two types of activities: one is the management of software components, including the specification, classification, and retrieval of existing components; the other is component integration that involves the integration of the reused components into an application. Over the past several years, a large number of techniques have been developed to address these reuse issues. However, the lack of a seamless integration of these techniques imposes significant obstacles to achieving effective reuse.

This paper surveys recent work based on the broad framework of software reusability research, and suggests directions for future research. We address general, technical, and non-technical issues of software reuse, and conclude that reuse needs to be viewed in the context of a total systems approach.

We begin with some basic definitions. Software reuse is the use of existing software or software knowledge to construct new software. Reusable assets can be either reusable software or software knowledge. Reusability is a property of a software asset that indicates its probability of reuse[17].

Software reuse purpose is to improve software quality and productivity. Reusability is one of the major software quality factors. Software reuse is of interest because people want to build systems that are bigger and more complex, more reliable, less expensive and that are delivered on time. They have found traditional software engineering methods inadequate, and feel that software reuse can provide a better way of doing software engineering.

A key idea in software reuse is domain engineering (product line engineering). The basic insight is that most software systems are not new. Rather they are variants of systems that have already been built. Most organizations build software systems within a few business lines, called domains, repeatedly building system variants within those domains. This insight can be leveraged to improve the quality and productivity of the soft ware production process [19]. The C++ language was also designed to encourage reuse as described in [1].

II. A FRAMEWORK FOR SOFTWARE REUSE

There are many approaches to the concept of software reuse. To organize and place various concepts and models of reuse (or reusability research), a number of conceptual frameworks for software reuse have been proposed.

A framework which classifies the available technologies for reusability into two major groups, composition technologies and generation technologies is proposed by Biggerstaff and Richter. Another framework based on three research and development questions, What is being reused? How should it be reused? What is needed to enable successful reuse? is developed by Freeman. In Freeman's framework, five levels of reusable information code fragments, logical structure, functional architecture, external knowledge (such as application domain knowledge and software development knowledge), and environmental knowledge related to organizational and psychological issues are defined. For each of the five information levels, typical projects of three different expected payoff periods are identified to answer research and development questions. Other frameworks by Horowitz and Munson and Jones are based on the forms of reuse such as data, code, and design.

III. SOFTWARE ARCHITECTURES

Software application architecture is the process of defining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security, and manageability. It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application.

Philippe Kruchten, Grady Booch, Kurt Bittner, and Rich Reitman derived and refined a definition of architecture based on work by Mary Shaw and David Garlan. Their definition is: "Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behavior as specified in collaboration among those elements; composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns."

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 [16]. Software architecture may be explored at different levels of abstraction. Shaw explored various structural models called architecture styles that were commonly used in software and then examined quality attributes related to each style. At a lower level of abstraction than style, [15] identified architectural patterns that commonly occur in various design problem domains such as client-server architectures, proxies, etc. In theory, these architecture patterns can be defined by applying a combination of architecture styles.

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 [19].

IV. SOFTWARE REUSE APPROACHES

The many different software development approaches can be separated into four categories: generation methods, composition methods, object-oriented methods, and the CASE approach[2].

4.1 GENERATION METHODS

The objects being reused are general problem solving patterns that drive the generation of the target programs. There are three classes of generation methods: language-based systems, application generators, and transformation-based systems.

4.2 COMPOSITION METHODS

Software development approaches that emphasize the composition approach utilize existing reusable resources that are viewed as atomic building blocks which are organized and combined according to well-defined rules. The major objective for these approaches is the creation of software libraries containing generic and reusable software components which can be combined to produce new target systems. This is the traditional view of reusability research. There are three areas of research emphasis: the development of application component libraries, the classification and retrieval strategies, and composition principles.

4.3 OBJECT-ORIENTED METHODS

Object-oriented programming languages provide another approach to reusability. A good discussion is contained in CACM. The properties of object oriented languages that help reusability include information hiding, property inheritance, and polymorphism. Information hiding is a reusability mechanism, since those parts of a system which cannot see information that must change can be reused to (re)build the system when that information does change. Property inheritance allows new subclasses to be built on top of super classes by inheriting variables and methods of the super class. The process of inheritance encourages reuse of previously defined data attributes and procedures in a more specific manner. Polymorphism means that operations have multiple meanings depending on the types of their arguments. Polymorphism can make reuse more flexible. Tarumi et al. have developed a programming environment for object-oriented programming which supports reuse of classes through the use of an expert system.

Object-oriented programming languages provide flexibility in using reusable objects. However, it is sometimes difficult to combine operations defined by different reusable objects. Even in an object oriented environment, a major problem is that it is still difficult for users, especially those who were not involved in the development of the existing software resources, to know whether there are reusable software resources to match their needs. Moreover, organizations will continue to use traditional software development approaches for reasons of inertia and efficiency as well as because of the large installed.

4.4 CASE TOOLS AND REUSE

Banker and Kauffman report that the level of code reuse is the major factor that deserves attention in software projects developed using CASE tools because extensive code reuse can increase productivity by an order of magnitude or more, and thus yield significant cost reductions in software development operations.

The central idea of CASE tools for reuse is the availability of software base containing software and software-related constructs such as domain knowledge and methodological knowledge. The availability of a software base makes application-oriented software reuse from early phases of the software development cycle (such as analysis and design) feasible with CASE tools. In contrast, most other current reuse approaches support only independent single component reuse at the coding phase.

Two different aspects of the CASE approach, integrated data dictionaries and code generators, are reported to promote software reusability by Oman. The data dictionary integrates all reusable software resources from various tasks into the central data dictionary and facilitates access to these resources for reuse purposes. CASE tools such as Excelerator and Prosa provide an integrated data dictionary. Code generators associated with a number of CASE tools automatically generate target source code from graphical software system models. CASE tools such as Cradle, HPS, IEF, IEW and Prosa have one or more code generators for programming languages such as Ada, C, COBOL, Pascal, and SQL.

V. SOFTWARE DESIGN PATTERN

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply object-orientation or more generally mutable state, are not as applicable in functional programming languages.

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.

There are many types of design patterns, for instance

- **Algorithm strategy patterns** addressing concerns related to high-level strategies describing how to exploit application characteristics on a computing platform.
- **Computational design patterns** addressing concerns related to key computation identification.
- **Execution patterns** that address concerns related to supporting application execution, including strategies in executing streams of tasks and building blocks to support task synchronization.
- **Implementation strategy patterns** addressing concerns related to implementing source code to support program organization, and the common data structures specific to parallel programming.
- **Structural design patterns** addressing concerns related to high-level structures of applications being developed.

VI. SOFTWARE REUSE: METRICS AND MODELS

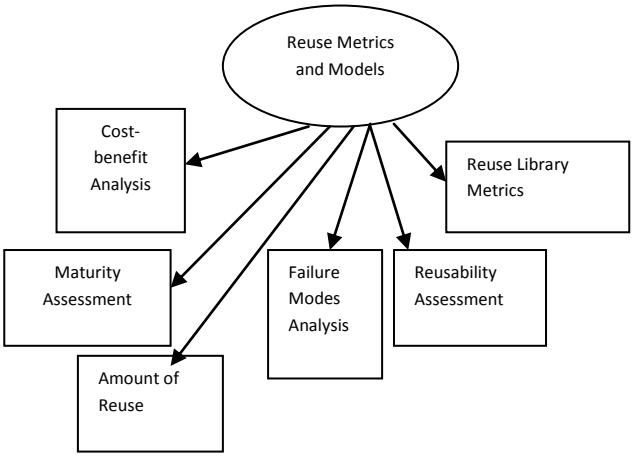

Fig 1.Software reuse metrics and models

As organizations implement systematic software reuse programs to improve productivity and quality, they must be able to measure their progress and identify the most effective reuse strategies. This is done with reuse metrics and models[4]. Figure 1, reuse models and metrics are categorized into types: (1) reuse cost-benefits models, (2) maturity assessment, (3) amount of reuse, (4) failure modes, (5) reusability, and (6) reuse library metrics.

- **Reuse cost-benefits models** include economic cost/benefit analysis as well as quality and productivity payoff.
  As organizations contemplate systematic software reuse, the first question that will arise will probably concern costs and benefits. Organizations will need to justify the cost and time involved in systematic reuse by estimating these costs and potential payoffs. Cost benefit analysis models include economic cost-benefit models and quality and productivity payoff analyses.
  Several reuse cost-benefit models have been reported. None of these models are derived from data, nor have they been validated with data. Instead, the models allow a user to simulate the tradeoffs between important economic parameters such as cost and productivity. These are estimated by setting arbitrary values for cost and productivity measures of systems without reuse, and then estimating these parameters for systems with reuse.
- **Maturity assessment models** categorize reuse programs by how advanced they are in implementing systematic reuse.
  Reuse maturity models support an assessment of how advanced reuse programs are in implementing systematic reuse, using an ordinal scale of reuse phases. They are similar to the Capability Maturity Model developed at the

Software Engineering Institute (SEI) at Carnegie Mellon University. A maturity model is at the core of planned reuse, helping organizations understand their past, current, and future goals for reuse activities. Several reuse maturity models have been developed and used, though they have not been validated.

- **Koltun and Hudson Reuse Maturity Model** To use this model, an organization will assess its reuse maturity before beginning a reuse improvement program by identifying its placement on each dimension. (In our experience, most organizations are between Initial/Chaotic and Monitored at the start of the program.) The organization will then use the model to guide activities that must be performed to achieve higher levels of reuse maturity. Once an organization achieves Ingrained reuse, reuse becomes part of the business routine and will no longer be
- **Failure modes analysis** is used to identify and order the impediments to reuse in a given organization.
  Implementing systematic reuse is difficult, involving both technical and non technical factors. Failure modes analysis provides an approach to measuring and improving a reuse process based on a model of the ways a reuse process can fail. The reuse failure modes model reported by Frakes and Fox can be used to evaluate the quality of a systematic reuse program, to determine reuse impediments in an organization and to devise an improvement strategy for a systematic reuse program.
  Given the many factors that may affect reuse success, how does an organization decide which ones to address in its reuse improvement program? This question can be answered by finding out why reuse is not taking place in the organization. This can be done by considering reuse failure mode hat is, the ways that reuse can fail.
- **Reusability metrics** indicate the likeli-hood that an artifact is reusable. Another important reuse measurement area concerns the estimation of reusability for a component. Such metrics are potentially useful in two key areas of reuse: reuse design and reengineering for reuse. The essential question is, are there measurable attributes of a component that indicate its potential reusability? If so, then these attributes will be goals for reuse design and reengineering. One of the difficulties in this area is that attributes of reusability are often specific to given types of reusable components, and to the languages in which they are implemented.
- **Reuse library metrics** are used to manage and track usage of a reuse repository. Organizations often encounter the need for these metrics and models in the order presented.

VII. REUSE LIBRARIES

Software Reuse Repository is simply a component library which stores the reusable components and must have the characterization of the assets that are included within. In order to make a effective use of a software repository, a reuser must have a clear understanding of its contents, so as to determine that whether his needs are likely to be met by the library. Repositories are used as mechanisms to store, search and retrieve components[6]. But finding and reusing appropriate software components is often very challenging particularly when faced with a large collection of components and little documentation about how they can and should be used. Many software component repositories have been developed often extending the approaches used for software libraries. The software reusable component is defined as "any component that is specifically developed to be used and is actually used in more than one context". This does not just include code, other products from the system lifecycle can also be reused such as specifications, requirements and designs. Components in this case can be taken to include all potentially reusable products of the system lifecycle including code, documentation, design, requirements, architecture etc. 'Development of software reusable repository' is required to implement a classification scheme to build a library and to provide an interface for browsing and retrieving components. The main requirement is to develop a classification scheme which is used for classifying components. The system should support three operations uploading components, downloading components and search for software components.

There has been disagreement in the reuse research community about the importance of libraries for reuse. However, failure modes analysis of the reuse process shows that in order to be reused a component must be available, findable and understandable. A reuse library supports all of these. The argument has also been made that most component collections are small and, therefore, do not need sophisticated library support. However, the emergence of the World Wide Web as a defacto standard library of reusable assets argues against this point of view [19].

Experiments on reuse libraries indicate that current methods of component representation could be improved. There is also a need for library environments that include facilities for configuration management and that integrate facilities for measurements such as usage and return on investment. The paper by de Jonge in this issue discusses how to handle the build process for reusable components.

VIII. COMPONENTRY

Component-based software engineering (CBSE) (also known as component-based development (CBD)) is a branch of software engineering that emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system[11]. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software. The broad interest in component-based software engineering has resulted in several component development, integration and deployment technologies. Most noted of these are Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) Component Model (CCM), Sun's Enterprise JavaBeans (EJB), and Microsoft's Component Object Model (COM+).

CORBA CCM allows integration and invocation of distributed components without concern for object location, programming language, operating system, communication protocol, or hardware platform. Concerns that cut across components, such as transaction handling, security, persistent state management, and event notification, are sup-ported by CORBA Object Services (COS).

EJB along with Java Remote Method Invocation (RMI) provides, as with CORBA, a platform for developing, integrating, and deploying distributed components. EJB provides an environment for handling complex features of distributed components such as transaction management, connection pooling, state management, and multithreading. This technology depends on the Java language but it achieves platform independence through the language. EJB, together with J2EE and Java servlets, provides a middleware platform for developing Web applications[16].

IX. DOMAIN ENGINEERING (PRODUCT LINE ENGINEERING)

Domain engineering, also called **product line engineering**, is the entire process of reusing domain knowledge in the production of new software systems. It is a key concept in systematic software reuse. A key idea in systematic software reuse is the application domain, a software area that contains systems sharing commonalities. Most organizations work in only a few domains. They repeatedly build similar systems within a given domain with variations to meet different customer needs. Rather than building each new system variant from scratch, significant savings may be achieved by reusing portions of previous systems in the domain to build new ones.

Domain engineering is designed to improve the quality of developed software products through reuse of software artifacts. Domain engineering shows that most developed software systems are not new systems but rather variants of other systems within the same field. As a result, through the use of domain engineering, businesses can maximize profits and reduce time-to-market by using the concepts and implementations from prior software systems and applying them to the target system. The reduction in cost is evident even during the implementation phase.

Domain engineering focuses on capturing knowledge gathered during the software engineering process. By developing reusable artifacts, components can be reused in new software systems at low cost and high quality. Because this applies to all phases of the software development cycle, domain engineering also focuses on the three primary phases: analysis, design, and implementation, paralleling application engineering. This produces not only a set of software implementation components relevant to the domain, but also reusable and configurable requirements and designs.

9.1 Family-Oriented Abstraction, Specification, and Translation (FAST).

Lucent Technologies introduced Family-Oriented Abstraction, Specification, and Translation (FAST) method in 1999 [7]. FAST applies product-line architecture principles into software engineering process. Thus, a common platform is specified to a family of software products. The platform is based on the similarities between several products close to each other. The variabilities among the members of a product family can be implemented with different variation techniques such as parameterization or conditional compilation.

The purpose of FAST is to make software engineering process more efficient by reducing multiple work, by decreasing production costs, and by shortening time-to-market. FAST process can be applied in a consistent and disciplined way. This is called PASTA (Process and Artifact State Transition Abstraction) model. PASTA model provides a path to follow during FAST process. It determines a set of steps that can succeed the current step. Thus, it gives precise instructions to follow, but still supports individual choices to make during the process. The purpose of PASTA is to make the software engineering process easy to iterate and reuse in future processes.

9.2 Domain Analysis and Reuse Environment(DARE), is a CASE tool that supports domain analysis – the activity of identifying and documenting the commonalities and variabilities in related software systems[5]. DARE supports the capture of domain information from experts, documents, and code in a domain. Captured domain information is stored in a domain book that will typically contain a generic architecture for the domain and domain-specific reusable components. The DARE process draws on three sources of information: code, documents, and expert knowledge as the basis for domain models. Information extracted from these three sources is used to build domain models such as facet tables and templates, feature tables, and generic architectures. All information and models are stored in a domain book. DARE has been used successfully in industry, for example, to support the building of text and database systems at Oracle [6].

9.3 Product Line UML-Based Software Engineering (PLUS). The field of software reuse has evolved from reuse of individual components toward large-scale reuse with software product lines. Software modeling approaches are now widely used in software development and have an important role to play in software product lines. Modern software modeling approaches, such as UML, provide greater insights into understanding and managing commonality and variability by modeling product lines from different perspectives[12].

The PLUS method extends the UML-based modeling methods that are used for single systems to address software product lines. With PLUS, the objective is to explicitly model the commonality and variability in a software product line. PLUS provides a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle software product lines.

The PLUS method is similar to other UML-based object-oriented methods when used for analyzing and modeling a single system. Its novelty, and where it differs from other methods, is the way it extends object-oriented methods to model product families. In particular, PLUS allows explicit modeling of the similarities and variations in a product line.

9.4 Feature-Oriented Reuse Method (FORM) was developed at Pohang University of Science and Technology (POSTECH) [8] and is an extension of the Feature-Oriented Domain Analysis (FODA) method [9]. It includes techniques and tools for product line engineering but has a loose process structure. This method has been applied to several industrial application domains, including electronic bulletin board systems, PBX, elevator control systems, yard inventory systems, and manufacturing process control systems, to create product line software engineering environments and software assets [10].

FORM is a systematic method that looks for and captures commonalties and variabilities of a product line in terms of "features." These analysis results are used to develop product line architectures and components. The model that captures the commonalties and variabilities is called a feature model. It is used to support both engineering of reusable product line assets and development of products using the assets.

9.5 Komponentbasierte Anwendungsentwicklung (KobrA). Fraunhofer
IESE has been developing the KobrA method, a component based product line engineering approach with UML and component-based application development method [11]. KobrA provides an approach to developing generic assets that can accommodate variations of a product line through framework engineering. The framework engineering starts with de-signing a context under which products of a product line will be used. The context includes information on the scope, commonality, and variability of the product line. Then, product line requirements are analyzed and the Komponent (i.e., KobrA component) specifications are developed. Based on the specifications, the Komponent realizations, which describe the design that satisfies the requirements, are developed. KobrA also provides a decision model that constrains the selection of variations for the valid configuration of products. KobrA includes both processes and techniques for product line engineering[17].

9.6 Koala, developed at Philips Corp. for analysis of embedded software in the domain of electrical home appliances, is an architecture description language [13] for product lines. In Koala, diversity interfaces and switches are provided for handling product variations. The diversity interfaces can be used to handle the internal diversity of components and the switch can be used to route connections between interfaces. When a component provides some extra functions, the access to these functions can be defined as optional interfaces. This enables the optimization of the code at compile time. Koala is a component-based product line engineering method with tools for integrating components both at compile-time and at runtime[17]. Koala is a descendant of Darwin [14] and is designed based on the experience of applying Darwin to television software systems.

X. BUSINESS AND FINANCE

The ultimate purpose of domain engineering and systematic software reuse is to improve the quality of the products and services that a company provides and, thereby, maximize profits. It is easy to lose sight of this goal when considering the technical challenges of software reuse and yet, software reuse will only succeed if it makes good business sense. Capital can be expended by an organization in many ways to maximize return to shareholders. Software reuse will only be chosen if a good case can be made that it is the best alternative choice for use of capital [19].
Business related reuse research has identified organizational structure to support corporate reuse programs, staged process models for reuse adoption, and models for estimating return on investment from a reuse program. More recent work has extended the return on investment analysis to include benefits from strategic market position [19].

Important problems remaining in this area include:

- Process focus.
- Sustaining reuse programs.
- Tech transfer.
- Reuse and corporate strategy.
- Organizational issues.

We will now discuss some of these issues.

10.1 PROCESS FOCUS

A software development process, also known as a software development lifecycle (SDLC), is a structure imposed on the development of a software product. Similar terms include software life cycle and software process. It is often considered a subset of systems development life cycle. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. Implementing a reuse program in a corporate environment requires a decision about when and where a capital investment is to be made. Development of reusable assets often requires a capital investment and there should be a strategic decision as to whether investment will be made proactively or reactively.
Proactive investment for software reuse is like the waterfall approach in conventional software engineering. The target domain or product line is analyzed, architectures for the domain are defined, and then reusable assets are designed and implemented taking foreseeable product variations into account. This approach tends to require a large upfront investment, and returns on investment can only be seen when products are developed and maintained. This approach might be suited to organizations that can predict their product line requirements well into the future and that have the [19].

Reactive investment is an incremental approach to asset building. One develops reusable assets as reuse opportunities arise while developing products. A sub domain with a clear problem boundary and projected requirements variations might be a good candidate. This approach is advantageous in that the asset development costs can be distributed over several products and no upfront large capital investment is necessary. However, if there is no sound architectural basis for the products in a domain, this approach can be costly as existing products may continuously have to be reengineered when assets are developed.

10.2 TECHNOLOGY TRANSFER

Here, we use the term "technology" to encompass a large number of things, and it is important for us to understand what "technologies" to study. For example, software engineers use a variety of techniques or methods to build and maintain software. We use the terms method or technique to mean a formal procedure for producing some result. By contrast, a "tool" is an instrument, language or automated system for accomplishing something in a better way. This better way can mean that the tool makes us more accurate, more efficient, or more productive, or that it enhances the quality of the resulting product. However, a tool is not always necessary for making something well.

10.3 ORGANIZATIONAL ISSUES

There are two types of commonly observed organizational approaches to establishing a reuse program: centralized and distributed asset development.

The centralized approach typically has an organizational unit dedicated to developing, distributing, maintaining, and, often, providing training about reusable assets. The unit has responsibilities to analyze commonalities and variability's of applications within the product line that have been developed or that will be developed in the future. The unit also develops standard architectures and reusable assets, and then makes them available to development projects. The unit maintains these assets and, often, also supports customization. The cost of this organizational unit is amortized across projects within the product line [19].

11. CONCLUSIONS &FUTURE RESEARCH

In this paper, I discussed different software reuse techniques and future scope for research. Software reuse is regarded as a key to improving software development productivity and quality. As outlined above, researchers and practitioners have proposed many approaches to increase the potential of software reusability. The full benefit of software reuse can only be achieved by systematic reuse that is conducted formally as an integral part of the software development cycle. Software reuse's safety and reliability issues are important and must be adequately addressed if reuse is to be a common practice.

There is a lot of scope for research in software reuse, like "Addressing the problem of increased maintenance costs", "Lack of tool support", "Reducing the cost for creating and maintaining a component library of software reuse", "Finding, understanding and adapting reusable components".

Currently, most reuse research focuses on creating and integrating adaptable components at development or at compile time. However, with the emergence of ubiquitous computing, reuse technologies that can support adaptation and reconfiguration of architectures and components at runtime are in demand. One implication of this development is that we somehow need to embed engineering know-how into code so it can be applied while an application is running. More research on self-adaptive software, reconfigurable context-sensitive software, and self-healing systems is needed.

REFERENCES

[1] B. Stroustrup, "Language-Technical Aspects of Reuse," Proc. Fourth Int'l Conf. Software Reuse (ICSR '96), 1996.

[2] Hafedh Mili, Patma Mili, and Ali Mili, "Reusing Software: Issuses and Research Direction", vol. 21, pp. 5-52, 1995.

[3] C. Krueger, "Eliminating the Adoption arrier," IEEE Software, pp. 29-31, July/Aug. 2002.

[4] W. Frakes and C. Terry, "Software Reuse: Metrics and Models," ACM Computing Surveys, vol. 28, pp. 415-435, 1996.

[5] W. Frakes, R. Prieto-Diaz, and C. Fox, "DARE: Domain Analysis and Reuse Environment," Annals of Software Eng., vol. 5, pp. 125-141, 1998.

[6] O. Alonso, "Generating Text Search Applications for Databases," IEEE Software, pp. 98-105, 2003.

[7] D.M. Weiss and C.T. R. Lai, Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, 1999.

[8] K.C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," IEEE Software, vol. 19, no. 4, pp. 58-65, July/ Aug. 2002.

[9] K.C. Kang et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI-90-TR-21, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, Penn., 1990.

[10] K.C. Kang et al., "Feature-Oriented Product Line Software Engineering: Principles and Guidelines, "Domain Oriented Systems Development: Perspectives and Practices K. Itoh et al., eds., pp. 29-46, 2003.

[11] C. Atkinson et al., Component-Based Product Line Engineering with UML. Addison- Wesley, 2002.

[12] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, 2004.

[13] R. Ommering et al., "The Koala Component Model for Consumer Electronics Software," Computer, vol. 33, no. 3, pp. 78-85, Mar. 2000.
[14] J. Kramer et al., "Software Architecture Description," Software Architecture for Product Families: Principles and Practice, M. Jazayeri et al., eds., pp. 31-64, 2000.
[15] F. Buschmann et al., Pattern-Oriented Software Architecture. Chichester, UK; New York: Wiley, 1996.
[16] W. Tracz, "DSSA (Domain-Specific Software Architecture) Peda-gogical Example," ACM SIGSOFT Software Eng. Notes, vol. 20, no. 3, pp. 49-62, July 1995.
[17] N. Leveson, Safeware: System Safety and Computers. Addison-Wesley, 1995.
[18] B. Meyer, ".NET is Coming," Computer, vol. 34, no. 8, pp. 92-97, Aug. 2001.
[19] William B. Frakes and Kyo Kang, " Software Reuse Research: Status and Future", vol. 31, no.7, pp.43-52, July 2005.

