Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 14 (20): 2396-2421, 2014
ISSN 1812-5654 / DOI: 10.3923/jas.2014.2396.2421

Taxonomy, Definition, Approaches, Benefits, Reusability Levels, Factors and Adaption of Software Reusability: A Review of the Research Literature

Ibraheem Y.Y. Ahmaro, Abdallah M. Abualkishik and Mohd Zaliman Mohd Yusoff
Department of Software Engineering, College of Information Technology,
Universiti Tenaga Nasional, Selangor, Malaysia

Abstract: Software reusability is an attribute that refers to the expected reuse potential of a software component. Software reuse not only improves productivity but also has a positive impact on the quality and maintainability of software products. The move toward reuse is becoming so widespread that it has even changed software industry's vocabulary. This study reviews the research literature on the concept of Software Reusability (SR). This study was conducted to provide a systematic review of the literature identify the definition, approaches, benefits, reusability levels, factors and adaption of software reusability. A systematic review was carried out of the research dealing with the content of software reusability, a literature search was conducted on several electronic databases. Studies published from the years 1977-2013 were considered and were selected if they described an evaluation of information and communication technology intervention to software reusability. In addition to that, a systematic review has been investigated on software reusability approaches and benefits. A deep investigation has been conducted on the definition, approaches, benefits, reusability levels, factors and adaption of software reusability. The concept of software reusability comprised of 11 approaches includes, design patterns, component-based development, application frameworks, legacy system wrapping, service-oriented systems, application product lines, COTS integration, program libraries, program generators, aspect-oriented software development and configurable vertical applications. Despite the rapid advancement in information and communication technology over the last decade, there is a limited evidence suggesting the adaption of software reusability. This study will help the information and communication technology industry to clarify how software reusability can benefit them by adapting the software reusability approaches.

Key words: Software reusability, design patterns, component-based development, application frameworks, legacy system wrapping, service-oriented systems, COTS integration

INTRODUCTION

Software penetrate in our daily life, possibly there is no other human-made material which is more omnipresent than software in our modern life. It has become essential part of many parts of society, shopping, telecommunications, home appliances, airplanes, personal entertainment, auditing, automobiles etc., In particular, technology and science demand high-quality software for making improvements and breakthroughs (Singh *et al*., 2010). Furthermore, Gill (2006) said the software development community is steadily moving toward the widespread software reuse where any software can be derived from the existing code. As a result, a growing number of software developers are using software not just as all-inclusive system, but also as module parts of bigger system. The reusability of code does not give the meaning that we will be able to copy-paste the same code in many parts in the system. In fact, it exactly means the opposite thing. Particularly, a piece of reusable code means that the same code can be reused in many parts without re-writing it (Singaravel *et al*., 2010).

Software reuse has been practiced since programming began. Reuse as a distinct field in software engineering, active areas of reuse research in the past twenty years comprise reuse libraries, reuse design, design patterns, component, generators, domain engineering methods and tools, measurement and experimentation, domain specific soft-ware architecture and business and finance. Significant ideas evolving from this period comprise systematic reuse, reuse design principles (Budhija and Ahuja, 2011).

The idea behind building reusable component is to design interchangeable parts from other industries to the

Corresponding Author: Ibraheem Y.Y. Ahmaro, Department of Software Engineering, College of Information Technology, Universiti Tenaga Nasional, Selangor, Malaysia

software field of construction. Building Reusable software components is newest tendency in the field of software construction. There are many work products that can be reused such as source code, specifications, designs, documentation and architectures (Jalender *et al*., 2012).

Software reusability is one approach that should be given due consideration for the benefits it brings, A good software reuse process is able to facilitate the increase of productivity of program design, development reliability of software product and the decrease of costs and implementation time (Burgin *et al*., 2004). Also reusability method is able to enhance the quality of the end product and also save time and cost (Sharma *et al*., 2009).

There is a limit evidence suggesting the adaption of software reusability in the industries, the industries fail to see how software reusability can benefit them in the long-term. Clearly there is a need to expand the software reusability concept so it will reach to more and more industries.

METHODOLOGY

A systematic literature search was conducted on eight databases. The databases were ACM Digital Library (DL), COMPENDEX, Emerald, IEEE Xplore Digital Library, ProQuest ERIC, Science Direct Backfiles Collections, Scopus and Springer Online Journal Collection, the search was performed using thesaurus terms and free text words, combining them in an appropriate way. The terms used were: software reusability, design patterns, component-based development, application frameworks, legacy system wrapping, service-oriented systems, application product lines, COTS integration, program libraries, program generators, aspect-oriented software development and configurable vertical applications. In addition, free text words were ANDed and ORed with the appropriate thesaurus terms and other search statements. English-language articles published in peer-reviewed journals from the year 1977 up to 2013 were included.

Iterative inclusion and exclusion selection criteria were developed and applied during two rounds of duplicate screening in our systematic review as shown in Table 1.

In the first round, titles and abstracts were screened by the author independently to retain articles featuring software reusability or any of its approaches. For the second round a screening of full-text articles were independently assessed and articles were selected. During the two rounds, the inclusion and exclusion were refined. Articles were excluded if they were out of our research scope. Publications that studied the same technology and approaches technology were grouped.

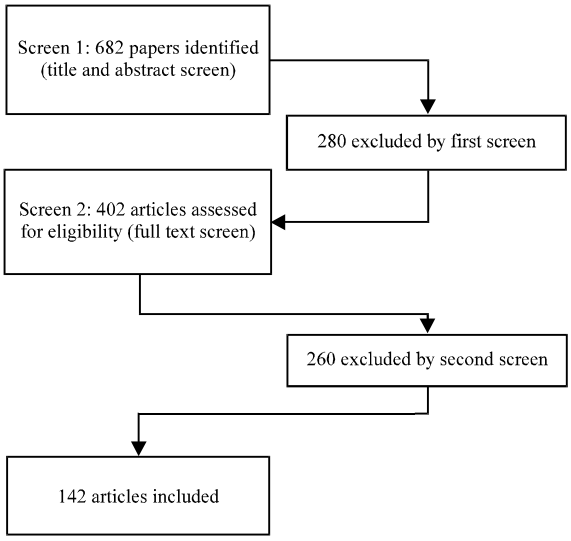

Fig. 1: Selection process for study inclusion

Table 1: Iterative inclusion and exclusion selection criteria

Round 1: Review of title and abstract
Inclusion criteria
Study generally the software reusability
And it's approaches
Round 2: Review of full text articles
Inclusion criteria
Must have an evaluation component (qualitative, quantitative or both)
Exclusion criteria
Focus on the definitions of software reusability
Focus on the approaches of software reusability
Focus on the benefits of software reusability

Initially the search yielded 682 papers. These papers were reviewed to exclude articles that did not meet the selection criteria: (1) Focus on the definitions of software reusability, (2) Focus on the approaches of software reusability, (3) Focus on the benefits of software reusability and (4) Paper written in English. Papers were retrieved for more detailed evaluation. The final number of papers included in the review was thus 142 as shown in Fig. 1. A total of 540 papers were excluded on the basis of the criteria specified in both rounds.

PRE-STUDY

Software reusability relates to using formerly written software in the form of specification, design and code. It is a practice which is widely observed in the process of development for most projects as it brings about several advantages.

Definition of software reusability: There are many definitions of software reusability. Software reusability is an attribute that refers to the expected reuse potential of

a software component (Kaur *et al*., 2012). Software reusability is defined as the degree to which a software module or other work product can be adapted in more than one software system or computer program (Fazal-e-Aminm *et al*., 2010). Jalender *et al*. (2012) claimed that software reuse is the use of engineering information or artifacts from existing software to build a new system, Software reuse is the use of existing software to construct new software. This is further emphasised by (Jalender *et al*., 2011) and (Singh *et al*., 2011) who said that software reuse is the process of building software systems from existing software rather than building them from scratch.

Kamalraj *et al*. (2009) said that the concept of software reusability can be reintroduced in new problem domain to reduce the required efforts and resources for constructing the software system. Furthermore, Rothenberger *et al*. (2003) mentioned that software reusability is a technique employed to address the required need for enhancing software development efficiency and quality.

According to Sharma *et al*. (2009) who defined software reusability as a tool to reduce development time and cost of the software. McCarey *et al*. (2008) reinforce this statement by saying that software reusability is also labeled as one which offers significant developments in software quality and productivity whilst reducing development costs.

Kim and Stohr (1998) stated that software reuse is the use of software assets from all phases of the software development in new applications and needs to be seen in the context of a total systems approach. Reusability is the likelihood a piece of source code that can be adapted and used again to add new functionalities with minor or no changes (Budhija and Ahuja, 2011).

According to Gill (2006) software reusability is an aspect that refers to the expected reusability potential of a software component. Moreover, reusability is considered as a form of usability. Software reuse is the process of updating or implementing computer programs or software systems using existing software assets (Burgin *et al*., 2004).

A study by Sandhu and Singh (2008) stated that software specialists have recognized reusability as a powerful technique of potentially overcoming software crisis apart from the significant improvements in software quality and productivity. Software reusability is an effective way of solving software crisis and improving software quality and productivity (Zhang *et al*., 2008). This is further emphasised by Sandhu *et al*. (2010) who said that software reusability is a primary attribute of software quality. Based on the definitions discussed above, Table 2 summarize the definition of software reusability.

SOFTWARE REUSABILITY APPROACHES

There are eleven software reusability approaches (Jalender *et al*., 2012; Kaur *et al*., 2012; Sommerville, 2004; Singh *et al*., 2010), which are design patterns, component-based development, application frameworks, Legacy system wrapping, Service-oriented systems, Application product lines, COTS integration, Program libraries, Program generators, aspect-oriented software development and configurable vertical applications.

Design patterns: In recent years, the influences of design patterns on the quality of softwares have attracted the increase of attention in the area of software development, design patterns encapsulate valuable information to solve design problems and to enhance the quality of the design. Design patterns are the amassed experiences by many programmers and developers to solve software design problems (Hsueh *et al*., 2011). Design patterns have proved as a significant piece that was needed from object-oriented design approaches (Al-tahat *et al*., 2001).

Singh *et al*. (2010) stated that, design pattern is a common reusable solution to a commonly arising problem in software design, design pattern is a description or pattern for how to solve a problem that can be adapted in several different circumstances.

Design pattern is a solution to the problem of a specific case. Design pattern is a record and refining of effective solutions which can be frequently repeated and

Table 2: Definition of software reusability

Software reusability definition	References
Is an attribute that refers to the expected reuse potential of a software component	Kaur *et al*. (2012)
Is the degree to which a software module or other work product can be adapted in more than one software system or computer program	Fazal-e-Aminm *et al*. (2010)
Is the use of engineering information or artifacts from existing software to build a new system	Jalender *et al*. (2012)
Is an aspect that refers to the expected reusability potential of a software component	Gill (2006)
Is the use of software assets from all phases of the software development in new applications and needs to be seen in the context of a total systems approach	Kim and Stohr (1998)
Is the use of existing software to construct new software	Jalender *et al*. (2011) and Singh *et al*. (2011)
Is the process of updating or implementing computer programs or software systems using existing software assets	Burgin *et al*. (2004)
Is an effective way of solving software crisis and improving software quality and productivity	Zhang *et al*. (2008)

Table 3: Main advantages for design patterns

Advantages	Description
Increased flexibility	Design pattern offers a structure to facilitate software modifications with respect to a design concern where it deliver a flexibility for potential changes
Increase the overall productivity	Design pattern enhance the documentation and maintenance of existing systems by furnishing an obvious specification of class and object interactions and their underlying intent
Reduce design problems	Design patterns help as tools to communicate ideas, solutions and knowledge about frequent recurring development design problems
Increase software reusability benefits	Design patterns increase advantages in terms of reuse make it easier to reuse successful designs and architectures

validated for the system developers. It mostly uses the basic thoughts of object-oriented and mechanisms to solve the recurring design problem of software and gives the reusable solution (weijiang *et al*., 2012).

A study by Palma *et al*. (2012) stated that each pattern describes a problem that occurs frequently in software development and then describes the core of the solution to that problem, inaway that we can use this solution several times. The use of design patterns deliver many advantages, from enhanced maintainability to enlarged reusability. Increasing number of available design patterns will mainly lead to better quality. Design patterns simplify the reuse of designs successful by offering solutions to software design problems, the adoption of design patterns has assisted in software reusability architectures. The significance of the reusability is due to the qualitative and economic advantages, it achieves (Al-tahat *et al*., 2001).

According to Aversano *et al*. (2007), design patterns boost advantages by reusing software which increases code quality, resilience to changes and maintainability. Bayley and Zhu (2010) reinforce this statement by saying that design patterns contribute to software reusability through enhancing concepts such as adequacy, expressiveness, readability and tool support. Table 3 summarize the main advantages for design patterns.

Component-based development: Component Based Software Development (CBSD) is an emerging technology that emphases on constructing systems by integrating existing software components. The CBSD offers a range of benefits includes: Enhance efficiency, enhance the ability to reuse components, managing growing complexity, reducing the time and effort needed to develop software, decreasing production costs through software reuse, enhancing the quality of the system, reducing maintenance costs, ensuring a greater degree of consistency, providing a wider variety of usability and supporting the effective use of specialists and increasing development productivity (Kahtan *et al*., 2012). This is further emphasised by Washizaki *et al*. (2003) who said, CBSD has become broadly accepted as a cost-effective method to software development, as it ensure the construction and design of software systems using reusable components. The CBD can reduce developmental process costs and enhancing the reliability of software system.

A study by Lingyun *et al*. (2010) stated that CBSD has been viewed the future trend in software development process due to not only saving the cost and time of system development, but also enhancing the reliability and maintainability of the system. It is thought to be one of the most significant methods to solve software development crisis. Modern software systems become large scale, complex and uneasily controlled, causing in high development cost, unmanageable software quality, low productivity and low maintainability.

Singh *et al*. (2011) said CBSD has been widely accepted and recognized in both industry and academia for constructing reusable components.

Hu *et al*. (2012) claimed that in CBSD, the system is not built from scratch, but developed from pre-existing components. Such development mode will improve the efficiency and reduce production cost with increased reuse. Zhang *et al*. (2012) affirmed this statement by saying that component-based development boosts a software development process that concentrates on component reuse. Gill and Tomar (2010) summarized the advantages of this approach as shown in Table 4.

Application frameworks: Application frameworks are widely used in order to boost efficiency and reliability in software development. An application framework is a reusable software product which delivers reusable design and implementation common to applications of a specific domain (Kume *et al*., 2012).

Frameworks represents the core of software development reuse methods, is the most proper solutions to simplify application development and overcome their development problems, using frameworks brings a many benefits to system development, enhancement of overall software quality and reduces development time and efforts (Al-Bashayreh *et al*., 2012a).

A study by Mailloux (2010) stated that application frameworks became a standard to implement and develop business systems. In most companies, application frameworks are central to most system developments

Table 4: Main advantages for component-based development

Advantage	Description
Reduced development time and cost	Reduced development process time by reusing existing software components, fewer programs are written and thus less time is spent in system developing
Improved quality	Enhanced quality because software components are often frequently reused, the defect fixes from each reuse accumulate.
Easily maintainable	Reusable components contribute to easy maintenance because they have fewer defects and they facilitate communication among software developers
Improved evolvability	Enhance the resolvability of software systems because it limit the needed change to components instead of identifying and changing all occurrences
Lower defect density	Reused components had a minor defect density than non-reused ones, with defect density calculated by dividing the number of defects by the number of lines of code

Table 5: Main advantages for application frameworks

Advantages	Description
Promote reusable design	Application frameworks enhance reusable design, modern development practices and tools
Reduces development cost	Application frameworks decrease development cost by the concept of modular and reusable programming
Enhance quality	Application frameworks improve system quality by 80% error reduction, 40-80% code reduction
Promote software reusability benefits	Application frameworks play an important part in software reusability since it is a set of reusable design and code that can assist in the development of software

because they help in facilitating or impairing the systems implementation. In most organizations, mastering the application framework allows greater developer productivity. Application frameworks can represent challenging design and conceptual work, attractive deep understanding of the technology, complex algorithm, integration with the operating system. Johnson (1997) reinforced about the importance of application frameworks and state that frameworks are becoming more important, citing systems like OLE, OpenDoc and DSOM. Table 5 summarize the main advantages for application frameworks.

Legacy system wrapping: By wrapping a set of defining interfaces by legacy systems offers access to interfaces. By rewriting a legacy system from scratch can create equal functionality information system based on modern software methods and hardware (Jalender *et al*., 2012). Legacy systems that can be wrapped by defining a set of interfaces and providing access to these legacy systems through these interfaces (Singh *et al*., 2010).

Accroding to Lee *et al*. (2001) software wrapping is cost effective and a short time solution at the lowest risk in system migration strategies. Wrapping is one of the most significant approach because it can preserved the merit of legacy systems which is reliable and stable at the minimum cost in short-term. Component wrapping method that reduces framework gap using instant framework between systems and legacy systems. The technique will assist in decrease the change for each of them, while increasing the reusability of the legacy systems.

Li and Qian (2009) claimed that adoption of wrapping technology for legacy system, stability of legacy system and original security are ensured in the integration platform. Zhang *et al*. (2008) mentioned that a legacy system is developed traditionally with a centralized, non-extensible architecture and mainframe-based, representing a massive, long-term business investment.

Li (2010) claimed that every legacy system is encapsulated as agent and these agents can cooperation with SOAP message simultaneously, we can integrate all agents distributed in different domains based on the integration aim. Due to the using of wrapping technology for legacy system, original stability and security of legacy system are ensured in the integration.

Souder and Mancoridis (1999) stated that the Legacy Wrapper is built to be a generic object wrapper, legacy systems offer services that stay useful beyond the means of the technology in which they were originally applied, the wrapper offers its own layer of security between the distributed object system and the security domains of the host.

The integration cost of legacy systems with modern technologies and systems decreased prominently, because the existing tools was adapted and the black box technique which does not contain the source code analysis is also adapted to uncover legacy functions to modern environments. Indeed low migration time and increases the performance of the generated Web Services (Parsa and Ghods, 2008). Table 6 summarize the main advantages for legacy system wrapping.

Service-oriented systems: In today's rapidly changing environment it is no longer possible for isolated systems to offer all the capabilities that are necessary to fulfill a mission. Service-orientation delivers unique benefits for heterogeneous and independently controlled systems (Simanta *et al*., 2010). Service-oriented systems are possess of distributable components using several technologies, working together to accomplish a common goal. Service-oriented systems are usually

Table 6: Main advantages for legacy system wrapping

Advantages	Description
Reduces framework gap	Using instant framework between systems and legacy systems
Offers access to interfaces	By rewriting a legacy system from scratch can create equal functionality information system based on modern software methods and hardware
Cost effective solution	reserved the merit of legacy systems which is reliable and stable at the minimum cost
Short time solution	Help to make the wrapping process automatic, reducing user intervention

Table 7: Main advantages for service oriented systems

Advantages	Description
More flexibility	Offer a more flexible method for software development
Better reuse	Provisioning and maintenance since services offer reusable functionality
Better placement of software systems with business and operations	Can be joint to support business processes that are dynamic in nature
Dynamic integration	Allowing software systems to be dynamically and reconfigured via services discoverable

deployed in networks of machines having several capabilities and assets (Van der Burg and Dolstra, 2011).

According to Carro *et al.* (2011) service-oriented systems have gained interest from industries and research communities internationally. This is further emphasised by Lewis *et al.* (2010a) who said, service-oriented systems represent a new class of systems, in which software is composed and adapted in the form of temporary services rather than being physically integrated, an essential attribute of service orientation is the using of services to rapidly improve and develop distributed applications.

Service orientation has been touted as the most significant technologies for developing, implementing and deploying major scale service provision system softwares. Service-oriented system is an approach for software development which services offer reusable functionality along with well-defined interfaces. Service-oriented systems consider as the approach that bridge the gap among software infrastructure and business models and flexibly supporting changing business needs. Decreasing the IT expenditures while rising the innovation potentiality through the investments in software (Kontogiannis *et al*., 2007).

Ling *et al.* (2010) claimed that service-oriented, provides a far less complex, scalable, more robust and faster way to develop, maintain, integrate and web-based software systems.

According to Simanta *et al.* (2010), service-oriented system is an architectural pattern for a software system of system (SoS) that includes: A set of services, that are operationally independent, autonomous and reusable components representing mission tasks, service users that can allocate and adapt services through standard, frequently published interfaces and an SOA infrastructure that link users to services, frequently through message-based communications approach. Table 7 summarize the main advantages for service-oriented systems.

Application product lines: Software Product Line (SPL) is a set of software-intensive systems sharing a similar, managed set of features that meet the particular needs of a specific market mission and that are constructed from a similar set of essential assets. Software Product Line (SPL) has proved very effective in developing large-scale software (Da Silva *et al*., 2010). Software product line (SPL) is a set of software systems which share a similar set of features and are constructed through similar set of core resources in order to enhance the efficiency of the product, reduce cost, time-to-market and boost software reusability, mass customization (Mohabbati *et al*., 2011).

Software product lines aim to attain the scope economies through the system development for products, diverse advantages such decreased time-to market, cost reduction and quality improvement which can be expected from reusing of domain-specific software assets. But also nontechnical advantages can be expected as consequence of network externalities, (Knauber and Succi, 2000). This is further explored by Carbon *et al.* (2008) who stated that Product line engineering has proven to be the best technique to optimize efficiency in producing sets of common software systems.

According to Hunt and McGregor (2006), software product line is a set of systems that share a managed, common set of features that meet the required market segment where APL is developed from a similar set of core assets in a prescribed way. Reusability based on software libraries has mainly peaked at about 30% of the content in products. Software product line is a recent reuse dimension which has delivered over 80% reuse and in several cases 100% reuse. Code is not only can be reused. Reuse starts in the initial stages of system development with creating the architecture, requirements and test plans.

According to Jalender *et al.* (2012), application product lines refer to techniques, methods and tools for creating a collection of product line systems from a set of software resources adapting a common models. An application type is generally around a common design so

Table 8: Main advantages for application product line

Advantages	Description
Mapping solution	Mapping from problem to solution domain can be easily described and automated by using the modelto- model transformations
More variability	Variability can be more concisely because next to the traditional mechanisms, variability can also described on the level of models
Modularization	Aspect-oriented methods allow the modularization and explicit expression of crosscutting variability on generator level, code and model
Traceability	Fine grained traceability is supported because tracing is done on the level of model element rather than on the code artifacts level

Table 9: Main advantages for COTS integration

Advantages	Description
Improve the quality	COTS based systems method is frequently considered significant for developing highly functional
Maintainability of systems	Larger granularity reusability and interface customization as the features of COTS relevant to their reuse
Flexibility of systems	Concentrate on a functional and non-functional to pick one or more products requiring to be modified for particular requirements and needs
Reduce the system development efforts	Reduced bugs thereby reducing the rollout the time and the risk system developments
Reduce the cost	Development cost is reduced by using the existing software assets and market proven software products

that it can be used in different ways for several different customers. A type of system that can be reused. Adaptation may include system configuration and component, select from an existing components from a library, modify components to meet new requirements or add the new components to the system. Singh *et al.* (2010) reinforced the concept by claiming that an application type is generally around a common design so that it can be used in different ways for many different customers.

A study by Voelter and Groher (2007) summarized the main advantages of product lines as shown in Table 8.

COTS integration: The software development world has rapidly evolved in the last decade. In particular, the adapting of commercial off-the-shelf (COTS) products as an essential elements of larger systems is becoming sharply commonplace, due to accelerating rates of COTS enhancement, shrinking budgets and expanding the requirement for systems. The term COTS is very generic, it can refer to several different types and software levels, for instance, software that is used as a tool to generate code software that offers a specific functionality (Morisio *et al*., 2000).

According to Raza *et al.* (2010), software in the current age are mainly developed by the integration of pre-fabricated COTS components because it is the easiest method to quickly develop systems that consume lesser cost than the traditional development methods.

Sanchez *et al.* (2012) stated that COTS software has present mainly in all engineering fields and certain COTS products such as AutoCAD and Excel are widely adapted. These products are developed for the use as stand-alone applications, their functionality makes them valid candidates for integration in big scale systems, where they can substantially enhance software reliability, decrease the system development efforts and improve systems quality. This is further emphasised by Kumar *et al.* (2010) who claimed that COTS integrations are getting widely recognized and adapted as one of the best methods for solving problems that required time to market solutions. The COTS integrations bring several benefits of reducing in house system developments, along with reduced bugs thereby reducing the rollout the time and the risk system developments.

A study by Singh *et al.* (2010) asserted that COTS technique supports systems to be built by integrating existing application systems and not from scratch. Wautelet *et al.* (2010) reinforced this statement by saying that many techniques have been recommended in literature to build COTS-based software systems.

In quality characteristics of COTS components are known and a model is presented that contain the quality factors, metrics and criteria. This model states mutual functionality, larger granularity reusability and interface customization (Fazal-e-Aminm *et al*., 2010).

COTS software in large-scale applications can be significantly beneficial as it reduces the cost of development, while increasing the overall quality (Ermagan *et al*., 2007). The use of COTS items can be benefits the implementation of new capability which can eliminate non-recurring costs that associated with systems development (Mott and Khan, 2000). Table 9 summarized the main advantages of COTS integration.

Program libraries: No one can doubt the importance of a well-stocked library of reliable routines to the successful of the practical computer system. Libraries are written as programs and involved in function literals, which are blocks that help as outer environments for other programs. Each library is conserved within some analyzer, which is a result of continuation from semantic suspension of analysis sometime modify the library itself has been analyzed. Some analyzer can be continuously form a complete user system or private or public libraries. (Wells *et al*., 1985). Program libraries considered at the top level of automation. Program libraries allow building the element by particular program according the specified commands (Parhomenko *et al*., 2003).

The structure, principles and design of an existing program library whose main purpose is to resolve the

optimize issues which are substantial discussed such a discussion to clarify the scope of system for potential users of the library and also is useful for workers on other software (Gill *et al*., 1979). Class and function of libraries employing a commonly used abstractions which are available for reuse. Libraries encompass code and data that offers essential services to independent programs. This idea boosts the sharing and exchanging of code and data (Jalender *et al*., 2012).

According to Gill *et al*. (1979), program library is a combination of routines that are written and envisaged within a consolidated framework, to be available to a broad community of users. Today there is a broad range recognition of the excessive value of program libraries developed to resolve useful classes of numerical issues, mathematical computing can thereby be made available to several users. The definition of a program library involves the existence of a set of purpose that exceeds the goal of any specific routine, the library should demonstrate a universal design that is reliable and steady with the presumption that the routines will be beneficial to a general-user community. Singh *et al*. (2010) claimed that function and class of libraries that implementing commonly-used abstractions are obtainable for reuse.

A study of commercial COBOL programs at the Raytheon Company revealed that several common functions at the company, and application stages were found to be reusable based on an examination of 5000 production, Raytheon designed a library involving 3200 reusable elements that includes a range of program logic structures and program categories. As a result, software programmers have been producing systems that on average includes a range of 15-85% reusable code (Cheng, 1994). Table 10 summarized the main advantages of program libraries.

Program generators: Program generator is a program that allows an individual to develop easily a program of their own with minimum programming and effort knowledge. With a program generator a developers may only be needed to state the phases or rules needed for his or her program and do not need to write any code, a generator system includes knowledge of a specific type of systems and can generate programs or system fragments in that domain. Program generators includes the reuse of algorithms and standard patterns (Jalender *et al*., 2012). This is further emphasised by Singh *et al*. (2010) who contend that generator program embeds knowledge of a specific types of application and be able to generate programs or system fragments in that realm.

A study by Markov *et al*. (2011) stated that program generation is the most promising and stable solutions for the software manufacturing dimension. Recent modifications in the way of developing software may lead to increase in the use of program generators to market extremely quickly.

Program generators can make the design very flexible. Rearrangement of the same meta-data will lead to changes in design. Anything which is repetitive can be automated. Given the right conditions for a particular situation, the majority of the code can be automatically generated. Therefore, the programmer is then free to fill in the gaps (Markov *et al*., 2011).

According to Sampath *et al*. (2007), program generators are programs that produce other programs. The input is a model in a particular modeling language and generate as output an implementation that captures the semantics execution. They play a significant part in addressing the increasing complexity of software engineering, program generators are also considered as a programs and it reflect the traditional methods for testing and measuring programs ought to be applicable to program generators. Smaragdakis *et al*. (2004) reinforced this statement by saying that program generation is one of the most methods in the effort to raise the automation of tasks in programming, program generators can be also made easy-to-implement.

A study by Markov *et al*. (2011) stated that some of advantages of automatically generated code are: Changes the way programmers work: Code tends to be much cleaner and simpler, good stability and low bug frequency, "works first time", produced very quickly, customizable and programmers are free to concentrate on the areas of development. Table 11 summarized the main advantages of program generators.

Aspect-oriented software development: In the last decade Aspect-Oriented Software Development (AOSD) has gained a wide-range of interest in both industry and academic institutions. The AOSD is an advanced approach for separation of concerns, which offers candid to modularize the crosscutting concerns and form it with

Table 10: Main advantages for program libraries

Advantages	Description
Improve the quality	Program libraries allow building the element by particular program according the specified commands
Reduce the system errors	Program library resolve the optimize issues which are substantial
Boost the reuse	Class and function of libraries employing a commonly used abstractions which are available for reuse
Boosts the sharing of code and data	Libraries encompass code and data that offers essential services to independent programs

Table 11: Main advantages for program generators

Advantages	Description
Reduce the development effort	Develop easily a program of their own with minimum programming and effort knowledge
Boosts the quality of the development	Most promising and stable solutions for the software manufacturing
Accelerate the development	Users are not required to know the language of programming that they are written on
Raise the automation	Software development automation of tasks in programming will be raised
Reduce the development cost	Provide space for the working thoughts of a specific problem domain without the burden of the cost

Table 12: Main advantages forAOSD

Advantages	Description
Variability	Variability is described in AOSD on model level thus it can be described more concisely
Automate the mapping	Mapping from problem to solution can be formally described and automated
Modularization	Enable the expression and modularization of crosscutting
Enhances the development mechanisms	Enhances the new composition mechanisms and the new aspect of abstraction
Increase the software quality	AOSD crosscutting concerns are Implemented and modeled independent of every one as well as the separation from the main functional concerns of the

the components of the system. Aspect-oriented software development usage has been increased in the past decade, AOSD development includes techniques as a means for the crosscutting concerns in systems development, in order to improving the return on investment and the development working practices in companies.

According to Jalender *et al*. (2012), aspect-oriented software development is a software development technology that endeavor new modularizations of systems to segregate the supporting functions from the core of the business logic program. Aspect-oriented software development allows multiple concerns to be expressed automatically and individually unified into the existing system working. This is further emphasised by Brichau *et al*. (2006), who claimed that AOSD is an emerging software engineering discipline that's transitioning from specialized industrial courses to postgraduate level teaching in universities.

According to Gerami and Ramsin (2011), AOSD has recently considered as a promising technique for modularizing and separating cross-cutting where object-oriented development implementation can result in scattering problems. This is further emphasised by Mehmood and Jawawi (2011) who state that AOSD environment crosscutting concerns are Implemented and modeled independent of every one as well as the separation from the main functional concerns of the system where this can be accomplished through the perceptions of Aspect-Oriented-Programming. AOSD has the potentiality to decrease the complexity of software. The AOSD aim is to provide a process for the systematic representation, separation, identification and composition of the crosscutting concerns (Budwell and Mitropoulos, 2008).

A study by Rashid *et al*. (2010) stated that AOSD mainly improve the design stability and substantial reduction in model size. Voelter and Groher (2007), reinforced this statement by saying that AOSD enhance the software development life-cycle by offer several ways for modularizing crosscutting concerns. AOSD are encapsulated as a powerful mechanisms shoring their compositional subsequent with other different software artefacts and can further increase productivity of software. Table 12 summarized the main advantages of aspect-oriented software development.

Configurable vertical applications: Configurable vertical application is a common system that is developed so that it can be configured to the requirements of particular customers. Sample of a vertical application is system that assists scientists manage their records, patient and insurance billing (Jalender *et al*., 2012). Vertical application considers as a tool which is concentrated on a narrow set of simulations (Ouyang *et al*., 2009).

Singh *et al*. (2010) contend that generic system is designed and developed for the purpose of meeting the system customers specific requirements by an routinely configured to meet the needs. Ouyang *et al*. (2009) further explored the configurable vertical application approach and stated that finite element modeling tools are subject to transformation in recently. These tools have been designed simpler to use, able to interact with outside applications and configurable for vertical applications. These softwares are configurable and anticipate the future needs of users, expected to interact with other software tools and efficiently create models. For the features that using the a well-featured and stable finite element modeling, some companies have designed a new software platform in order to develop finite element modeling software that can meets the encountered issues for today's users, both from a data management and infrastructure standpoint. Mainly this provide the users with an opportunity to develop, manage and share models by adapting scripting tools and standardized interfaces. The transformation promising of the development process and product design can be achieved by combining a various vertical applications where it help to leverage

Table 13: Main advantages forconfigurable vertical application

Advantages	Description
Facilitate the configuration	Is a common system that is developed so that it can be configured to the requirements of particular customers
Boost the development	Is a tool which is concentrated on a narrow set of simulations
Help in the transformation	Transformation promising of the development process and product design can be achieved by combining a various vertical applications
Increase the anticipate	Anticipate the future needs of users, expected to interact with other software tools and efficiently create models

optimization tools, automated modeling and simulation to identify optima design. Table 13 summarized the main advantages of configurable vertical application.

BENEFITS OF SOFTWARE REUSABILITY

Budhija and Ahuja (2011) and Singh *et al.* (2010) have agreed that the major benefits of software reusability are: Increase dependability, increase productivity, increase effectiveness, accelerate development and cut operational costs.

Increased dependability: Budhija and Ahuja (2011) and Singh *et al.* (2010) have agreed that reused software, should be more dependable than new software because it has been tested and tried in working systems. The initial use of the software reveals any faults in the design and implementation phases, these are then fixed, consequently reducing the number of failures when the software is reused.

According to Sandhu *et al.* (2010), increasing the dependability will reduce the time of the software development since it minimize the development failures. Sommerville (2004), reinforced this statement by saying that reused software that has been scabbed and tested in functioning existing systems is more dependable in compare to new systems because the number of failures is decrease and the errors are minimal.

Increased productivity: Software reusability improves portability, productivity and quality of software (Sagar *et al.*, 2010). A good software reuse assists the increasing of reliability, quality and productivity (Jalender *et al.*, 2011). Software reusability enhances productivity and also has a positive influence on the maintainability and quality of software products (Gill, 2006).

According to Singh *et al.* (2011), the concept of reusing for an existing software components consider as a key feature in increasing productivity. McCarey *et al.* (2008) reinforced this statement by saying that software reusability provides important developments in software quality and productivity whereas reducing development costs.

Increased effectiveness: A successful program of software reuse provides several advantages among others: Increase in the overall effectiveness of the software development process. The increase of software quality with reuse comes along because reusable components have been tested and over time, become almost errors-free (Kim and Stohr, 1998). Instead of systems developers doing the same work on several different projects, these mechanism can create, design and develop reusable software that encapsulate their knowledge (Singh *et al.*, 2010) and (Budhija and Ahuja, 2011).

A study by Sandhu *et al.* (2010) stated that software reusability provide an effective use of standards compliance, specialists and accelerated the software development processes. Reused software components that have been tested and tried reduce the margin of error in project cost estimation and it is more effective than new software.

Accelerated development: Reusing software can accelerate the production of system because both validation and development time should be reduced. Mainly bringing a system to market as early as possible is more important than overall development costs (Singh *et al.*, 2010).

According to Jalender *et al.* (2012), one of the main benefit of the developing reusable software components is that it reduces the time in developing any software. This is further explored by Lemley and O'Brien (1996) who claimed that the idea of software reuse includes software engineering integrating from existing systems into the developing of new software which will lead to a reduction in the developing time of new software system.

Reduce operational costs: If software exists, there is less doubt in the reusing costs of software than the costs of development from the scratch. This is significant factor for any project as it decreases the error's number in project cost estimation. This is mainly true when a large software components are reused (Singh *et al.*, 2010; Budhija and Ahuja, 2011).

Table 14: Software reusability and its main factors

Factor name	Representative quote
Flexibility	Flexibility is the ability to use it in multiple configurations, To reuse some source code component it should be flexible to be used in many contexts. Flexibility is important for the required changes with the passage of time, it saves developers not to be bound
Maintainability	Maintainability is a major issue, when you use OSS and we are running the system with connectivity with other different systems, the new bugs and removing the bugs in source code that is designed by some other is very hard for developer
Portability	Portability is refers to install ability, portability should be economical, where we don't need to install any other softwares in other systems to run the components
Scope coverage	Developers have to choose normally the more coverage component as compare to the less covered one, this is further depends on the application if developers want to extend more the application then they will go for additional features
Stability	Stable means the reasonability error is free and it may be adapted with confidence that there is no bug
Understandability	If developers do not understand it then developers can't show that it is reliable and prove it to be useful, The size could be managed but if it is not understandable then it is hard to reuse
Usage history	Usage history shows the component's maturity and how often people have adapted and made changes to it, Open source software is used by several folks and engineers and already proven its usefulness
Variability	Variability is a two edge sword in other words there are advantages and disadvantages
Documentation	Any problem with the documentation then It means it has creates hurdles to understand the code for any other software engineer or developers, If there any problem with the documentation then other developers cannot understand the software neither can change nor modify it

According to Singh *et al*. (2011), software reuse has been considered as a tool to reduce the development cost, the cost of the system developing from scratch can be saved by extracting and identifying the reusable components from previously developed systems or legacy systems.

REUSABILITY LEVELS IN SOFTWARE LIFE-CYCLE

Reusability have few levels include: Code reuse, design reuse, specification reuse and application system reuse (Cheng, 1994).

Code reuse: Code reuse consider as the most common form of software reuse. This type of reuse happens in the development implementation stage of the system development process. The reusable code can be object code, data objects, source code and standard subroutines.

Design reuse: Design reuse considers as a higher level of reuse where a design model of a software system is reused. This type of reuse is required when a system needs to be reported in an entirely different software or hardware environment.

Specification reuse: The reuse of specifications consider as a higher level of reuse, the problems at the specification level which are arising from the inefficiencies of reusable code, operating system dependencies and programming language dependencies disappear.

Application system reuse: Application system reuse is consider as a special case of software reuse, where the whole system is reused by implementing it through a range of different operating systems and computers.

FACTORS AFFECTING SOFTWARE REUSABILITY

Software reusability is the characteristic and attribute which relate to the potentiality of software to be reused, primarily there are 9 factors that involved with the software reusability assessment includes: Flexibility, Maintainability, Portability, Scope Coverage, Stability, Understandability, Usage History, Variability and Documentation (Fazal-e-Amin *et al*., 2011) as shown in Table 14.

ADAPTION OF SOFTWARE REUSABILITY IN INDUSTRY

Software reusability and its related artifacts provides the potential cost savings in several industries, the cyber infrastructure development that is used by the earth science have been effected by the software reusability artifacts. The software reusability value can be readily noticed by the growing of the open source community of reusers and developers of software, which has been modifying the system development practices in many industries. Likewise, the community of earth science systems (Marshall and Downs, 2008).

Accroding to Sagar *et al*. (2010), developing systems with reusable components bring several benefits to organizations. There are several direct or indirect factors as if efforts, time and cost. A reusable component helps in low maintenance efforts and better understanding for the application.

A study by Jalender *et al*. (2012) stated that some quantitative benefits achieved through adapting software reusability in industry which are presented in Table 15.

Table 15: Software reusability in industry

Organization	Representative quote
Nippon electric company	Achieved a higher productivity by 6.7 times, better quality by 2.8 times through 17% reuse and enhanced software quality 5-10 times
GTE corporation	Saved $14 million in costs of software development with reuse levels of 14% GTE. For 20-50% reuse in 1988 GTE Data services benefited from $1.5M in cost savings
Toshiba	With reuse levels of 60% there was a 20-30% reduction in defects per line of code
DEC	With the reuse levels of 50-80% DEC reported cycle times that were reduced by a factor of 3-5 and an increase of 25% in productivity
Hewlett-Packard (HP)	With reuse levels up to 70%. HP saw a quality improvement on two projects of 76 and 24% defect reduction, 43% reduction in time to market and 50 and 40% increases in productivity
Raytheon	With a 60% reuse using COBO there was a 50% productivity increase in the MIS domain
312 projects in aerospace industry	Showed an average 20% increase in productivity, 25% reduction in time to repair, 20% reduction in customer complaints and 25% reduction in time to produce the system
NASA	Report that in overall development effort and cost there is a reduction of 75%
AT and T	Reported that with a 40-90% reuse there is a 50% decrease in time-to-market
SofTech	Reported that with the reuse greater than 75% there should be a 10-to-20 times increase in productivity
U.S. Department of Defense	Increased the level of rcuse by as little as 1% and saved 300 million annually
QSM associates Inc	Reported that component assembly helps in reducing the life-cycle of software development, 84% reduction in project cost and a productivity index of 26.2, compared to an industry norm of 16.9
University of California	Reported that an increase in productivity by 20% through a 62% software reuse level in building prototypes
Japanese software factories	Reported that there was an increase in software productivity eight times through a 85% software reuse level
NASA	Reported that a study found that 42 percent of their third Ada project code was reused code

DISCUSSION

According to Jalender *et al.* (2012), there are several technical issues that currently keep reusable software from becoming a reality. One of the techniques is designing code level reusable components. In this approach the technical issue is the lack of formal specifications for components. A programmer cannot be expected to reuse an existing part unless its functionality is crystal-clear. A component will only be reused if its behavior is completely and unambiguously specified in a form understandable by potential programmers. These specifications should be mathematically rigorous. Specifically, informal natural language descriptions are not sufficient.

According to Singh *et al.* (2010), a chunk of code is regularly organized using modules or namespaces into layers. 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 or scores of reusability. Some problematic issues that needs to be addressed related to systematic software reuse are: A clear and well-defined product vision is an essential foundation to an software product lines, An evolutionary implementation strategy would be a more pragmatic strategy for the company, there exist a need for continuous management support and leadership to ensure success, an appropriate organizational structure is needed to support SPL engineering and the change of mindset from a project-centric company to a product-oriented company is essential.

Fazal-e-Amin *et al.* (2011) stated the objectives of reusing software are to reduce the cost and amount of resources used to produce quality software that is on time. These objectives are achieved by reusing software artefacts. The reuse insensitive software development approaches, such as Component Based Software Development (CBSD) and Software Product Lines (SPL) development, make use of reusable software assets.

Cheng (1994) described that software reuse has had only limited success in practice. This is because that there are non-technical factors as well technical ones affecting software reusability.

According to Sandhu *et al.* (2010), software reusability is primary attribute of software quality. In the literature, there are metrics for identifying the quality of reusable components but there is very less work on the framework that makes use of these metrics to find reusability of software components.

Fazal-e-Aminm *et al.* (2010) stated that the software engineering research community has made enormous contributions in past decades to lay down its foundations. However, the research methodologies in software engineering are still not mature. The focus recently is on the aspect of the evaluation of reusability assessment approaches.

According to Kaur *et al.* (2012), the reusability is the quality of a piece of software, that enables it to be used again, be it partial, modified or complete. Software professionals have recognized reuse as a powerful means to potentially overcome the situation called as software crisis.

Sharma *et al.* (2009), proposed a Artificial Neural Network based approach to assess the reusability of software component has been presented to help developers to select the best component in terms of its reusability, which will improve the maintainability of the overall system and will also provides some guidelines

to augment the level of software reusability in component-based development, which is summarized as the following: Conducting thorough and detailed software reuse assessment to measure the potential for practicing reuse, performing cost-benefit analysis to decide whether or not reuse, adoption of standards for components to facilitate a better and faster understanding of a component, selecting pilot projects for wider development of reuse and identifying reuse metrics.

According to Budhija and Ahuja (2011), 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.

Singh *et al.* (2011), proposed a model based on four parameters: Changeability, Interface Complexity, Understandability of Software and Documentation Quality for accessing software reusability levels using soft computing techniques viz., Fuzzy Logic, Neural Network and Neuro-Fuzzy. The proposed model using Neuro-Fuzzy technique is trained well and predicts satisfactory results.

Burgin *et al.* (2004) stated the development of a methodology and mathematical theory of software metrics for evaluation of software reusability allows one to use experience in the development and utilization of software usability metric for the development and utilization of software reuse metric. Different types and classes of software metrics are explicated and compared.

ISSUES AND CHALLENGES

De Oliveira *et al.* (2009) stated the recent advances in technology and platforms, which occur in new computing areas have demanded a great effort of Software Engineering to offer new approaches, which bear the software development with quality and lower cost. The growing trend toward systems configured of individual components has taken the original concept of reuse into a completely different arena. It has also presented many challenges to software developers attempting to enter this new arena (Morisio *et al.*, 2000). Organizations, having large-scale and complex applications development, still face a lot of problems, especially while testing and updating the systems. So until systems are not designed carefully, they may be costly Raza *et al.* (2010).

According to Mohabbati *et al.* (2011), real industrial feature models tend to grow very large up to thousands of features. Configuration of such large feature models is very costly and overwhelming for application developers. This process tends to be even harder in the case of multiple stakeholders, when they have different priorities for high level non-functional properties. With the advance of the Cloud Computing paradigm, new challenges in terms of models, tools and techniques to support developers to design, build and deploy complex software systems that make full use of the cloud technology arise (Cavalcante *et al.*, 2012).

Li and Qian (2009) said that nowadays there are many information systems running in some enterprises for different purposes, but on the other hand, it is very difficult to implement cooperation among these systems. These systems were developed at different time by different vendors, their functions are isolated and inconsistent with current technology and/or protocol and these different systems may be instituted on different operation systems and different areas; all these bring up the problems so called "Information Island".

Washizaki *et al.* (2003) stated that it is necessary to measure the reusability of components in order to realize the reuse of components effectively. However, in application development with reuse, it is difficult to use conventional metrics because the source codes of components cannot be obtained and these metrics require analysis of source codes.

In this section, we attempt to identify the remaining challenges to be addressed and the promising research avenues in the domain of software reusability. The main issues and challenges for the software reusability approaches have been discussed and listed down as the following.

Design patterns: According to Hsueh *et al.* (2011), design patterns are the accumulated experiences by many developers to solve specific software design problems. Developers follow the guidelines to master their design style and avoid design abuse. the developers may misuse the design pattern since the pattern applied in the original design does not contribute to the subsequence design, experienced developers apply design patterns in software development to solve design problems and reduce software maintenance cost. However, software systems evolve over time, increasing the chance that the design patterns in its original form will be broken. However, although the design pattern embraces the property of extensibility, it may not exert the effectiveness in a software evolution. Software design patterns are best practice solutions to common software design problems (Fant *et al.*, 2011). A pattern describes a solution tem-plate to a recurring problem encountered in a specific context as well as consequences of using the template and of possible choices in adapting the template Stepan (2011).

Weijiang *et al*. (2012) reported the first problem for the developers to solve, is the user interface design. In general, the user interface is very easy to change, therefore, it is necessary for the designer to separate user interface and function of the system to make them independent and allow the flexibility to change user interface without affecting the functional part, design patterns greatly improve the scalability, reusability and maintainability of the software and to effectively face the challenge of changing in future. Composite pattern is used to solve the problem of rendering complex graphics, Bridge pattern is used to solve the problem of calling different calculation algorithm based on multiple data sources, Command pattern is used to solve the undo/redo problem. Pattern detection is an important part of many solutions to Software Reuse practices Gupta *et al*. (2011).

According to Aoyama (2000), design patterns are a set of mapping from a problem space to a solution space, problem space is not a single space but is composed of multiple problem spaces. It should be noted that design is based on the decomposition of problem space, instead of solution space. Since design activities start in problem space, it's desirable to navigate designer in problem space. Design patterns refer to reusable or repeatable solutions that aim to solve similar design problems during development process. Various design patterns are available to support development process. Each pattern provides the main solution for particular problem. Most developers lack of knowledge on existing design patterns. The existing and well proven design experiences help us in finding appropriate solutions to design problems Thung *et al*. (2010).

According to Jensen and Cheng (2010), design patterns provide a context-driven solution template for solving design problems that occur frequently in large software projects. However, using metrics and design patterns often requires significant intellectual investment, thus limiting their adoption. Gamma design patterns addresses problems related to classes and their associations that make up the structure of an object-oriented software design. Maintaining an object-oriented design for a piece of software is a difficult, time-consuming task. Designing that software to be easily maintained and extended in order to satisfy new requirements is even more difficult, as it forces developers to consider not only the details of the solution but also details of the problem space. These difficulties led to the development of two important enabling technologies in software engineering. It is difficult to follow all new patterns during development and to choose the right patterns when faced with a design problem. Design Pattern Recommender aimed to help designers to find or decide which pattern to use for an articular design problem Palma *et al*. (2012).

Loo and Lee (2010) stated that design patterns are known as a way for software designers to communicate about design. There are various descriptions, structures and behaviors on the solution for a design problem in a design pattern. However, there is not much visual aid on the internal workings of a design pattern in a visual design modeling tool. Currently, it is difficult to determine the pattern role and the variant of interaction groups of a design pattern in an UML diagram as the design pattern information is not represented in the interaction diagram. There is a need to have a consistent way to define the pattern role participating in a design pattern interaction and whether there is a variant in each interaction group.

Ramirez and Cheng (2010) studied harvesting design patterns is a difficult and subjective process because there is no standard methodology for developing design patterns in practice. no metrics are available to quantify the quality of a resulting design pattern. Architectural patterns are general reusable solutions to commonly occurring problems in software design. They offer well-established solutions to architectural problems, help to document the architectural design decisions and facilitate communication between developers through a common vocabulary (Vuksanovic and Sudarevic, 2011).

Component-based development: According to Zhou *et al*. (2011), CBD is such a paradigm is not fully exploited by many software companies because of the much needed effort and cost. The CBD approach is more than a technological problem, we believe a lightweight approach can address many problems and also influence other related issues to make a life easier, in real practice, such a paradigm is not fully exploited by many Enterprise Application (EA) software companies because of the much needed effort and cost. The CBSD purports to address the problem of systems which are delivered behind schedule, over-budget and inadequately meeting user requirements Sharp and Ryan (2010).

Basha and Moiz (2012) stated that though there is a rich set of software metrics available but there is a dire need for research in developing the component metrics as the program volume measure, potential volume measure, development of domain specific components and its impact on effort in terms of cost and time is still a challenging issue. The disjoint processes of software components productivity and grouping will posses a huge significant problem for all the stakeholders in the software components marketplaces. Much research needs to be carried out in this domain in order to provide more secured and efficient collection of software components. Standardization of component development process is a challenging issue. There is a need for unified component development and unified component testing process.

According to Panwar and Tomar (2011), software crisis is very big problem to maintain the software quality and to handle this problem. The CBSE is very helpful to improve the productivity. But to maintain the quality it is necessary to measure the attributes of the software to make it reliable and reusable. In CBSE, to make reliable and reusable software it is very necessary to check the quality at every phase before testing to detect the faults at early stage of software development lifecycle. The use of conventional metrics in CBD is difficult, because these metrics needs analysis of source codes. To assess the reuse of component, it is important to estimate reusability of these components Sagar *et al.* (2010).

Pande *et al.* (2013) stated that software systems are complex and may involve large numbers of components and requirements, a component selection approach is needed that is also capable of addressing large-scale problems. The CBD lacks appropriate reusability guidelines that could further benefit component-based development from cost-savings, time-savings, quality and productivity improvements, reliability improvements, one of the key challenges faced by software developers is to make (CBD) an efficient and effective approach. Since components are to be reused across various products and product-families, components must be characterized and tested properly Gill (2006).

According to Kahtan *et al.* (2012), the central problem with CBSD is the difficulty in ensuring reliability and other non-functional requirements of the components and thus the inability to ensure that specific application attributes are secure. security features of software components must be considered and evaluated earlier in the CBSD lifecycle.

Zhang *et al.* (2012) said that components should be described using three different forms at three development stages: architecture specification, configuration and assembly. However, no architecture description language proposes such a detailed description for components that supports such a three step component-based development.

Crnkovic (2012) said that the software development requires approaches that can manage that complexity in a similar way as this is done in general-purpose software, but at the same time provide support for embedded systems specifics. While bibliographic data from most of the libraries was transferred relatively straightforwardly, the transfer of administrative data represented a greater challenge since it is not standardized and varied greatly between existing systems (Vuksanovic and Sudarevic, 2011).

Application frameworks: According to Mailloux (2010), application frameworks became a standard to implement and develop business systems. Application frameworks can represent challenging design and conceptual work; attractive deep understanding of the technology, complex algorithm, integration with the operating system.

Lee *et al.* (2001) stated that multiple frameworks are composed, there are many problems that are framework gap, overlap of framework entity and framework control etc. The causes of these problems are related to the cohesion of each frameworks, the scarcity of the understanding of the programs and design, the lack of access to source code of the framework in object oriented framework. Software framework provides software libraries that offer solutions for most common programming problems, with the goal of eliminating repetitive operations (Vuksanovic and Sudarevic, 2011).

According to Al-Bashayreh *et al.* (2012b), frameworks represents the core of software development reuse methods, is the most proper solutions to simplify application development and overcome their development problems, The software frameworks achieve the concept of inversion of control by encapsulating both control flows and object interfaces. Moreover, software frameworks are more specialized for a particular problem, while libraries are more general. Application framework allows the system developer to produce complex applications with a modest development effort and also application framework provides reusable solutions for difficult, recurring problems (Bohn *et al.*, 2008; Rittammanart *et al.*, 2008).

Legacy system wrapping: Zhang *et al.* (2008) stated the major incompatibility of legacy systems lies in their obsolete non-extensible software architecture that has hampered legacy systems from modernization evolvements. Many legacy systems are non-decomposable as black boxes to users for commercial consideration, making the modernization task even more challenging. A significant problem in the modernization of legacy systems is how to maintain interoperations between users and legacy interfaces that are required to be reengineered and deployed in the Internet. The problem of Legacy System Modernization is not novel in the literature, where a number of approaches can be found. A taxonomy of Legacy System Modernization techniques distinguishing between redevelopment, wrapping and migration techniques which redevelopment is unacceptably risky and wrapping is unsuitable, while migration is a feasible option.

Li (2010) claimed that every legacy system is encapsulated as agent and these agents can cooperation with SOAP message simultaneously, we can integrate all agents distributed in different domains based on the integration aim. Due to the using of wrapping technology for legacy system, original stability and security of legacy

system are ensured in the integration. Reengineering a legacy system to provide Web Services is a great challenge. Wrapping legacy systems is a proper solution to expose legacy program functionalities as Web Services. the main challenge is to remove the burden of handing code custom wrappers and connectors for legacy programs (Parsa and Ghods, 2008).

Chenghao *et al.* (2010) claimed that the legacy systems are becoming incompatible with service-oriented computing, it is a challenging task for migration of component based legacy systems towards service oriented applications, due to non-extensibility, non-decomposability and a high coupling degree of legacy system, the problem is arising when migrated legacy systems to service-oriented systems. After a decision that ensures what can be migrated form the original legacy system is made, the technical problems that how the migration can be executed is concerned. Legacy systems are those whose usefulness has extended beyond the expectations of their creators. The Legacy Wrapper attempts to extend the usefulness of those applications by facilitating their integration into modern distributed systems Souder and Mancoridis (1999).

Service-oriented systems: Khazankin *et al.* (2011) stated that the service-oriented systems have become an important approach and technological framework to solve problems in distributed computing environments. Challenges in distributed service-oriented systems include the discovery of resources and monitoring of the system's runtime behavior, adaptive request scheduling in such systems is challenging due to the poorly predictable behavior of human actors in performing tasks. The engineering of service-based applications represents a significant new challenge for computer science, to create communities of services that are always connected, frequently changing, open or semi-open and form the baseline environment for software applications. However, this shift brings about not only potential benefits, but also serious challenges for how such systems and applications should be designed, managed and deployed (Vazquez-Salceda *et al.*, 2010).

According to Newman and Kotonya (2011), there is lack of research initiatives investigating the negative impact resource contention can have on resource constrained systems and the dynamic reconfiguration of executing services to mitigate this resource contention. Suring quality is particularly problematic for service oriented systems which operate in resource-constrained environments. Technologies that support service orientation are not enough, or where the investment is too large to justify its implementation Kontogiannis *et al.* (2007).

Ling *et al.* (2010) said that in today's business-critical environments, there is only a limited possibility that all services are to be developed from scratch. To reuse prior knowledge of existing object oriented system design and adapt them to more flexible and scalable service-oriented systems. Service-oriented systems is representing a new class of systems, Service-oriented systems have attracted great interest from industry and research communities worldwide. Service integrators, developers and providers are collaborating to address the various challenges in the field. A multitude of approaches and tools have been proposed to support different areas of service oriented systems such as service description, service design and development, service discovery, service composition, service adaptation and service management and monitoring Lewis *et al.* (2010b).

Nitto (2009) reported that the service-oriented paradigm is emerging as a new way to engineer applications that are exposed as services for possible use through standardized protocols. Service-oriented applications are pushing traditional software engineering problems-distribution, componentization, composition, requirements, specification, verification and evolution-to their extreme. Service-oriented architecture is a relatively new approach to software system development. It divides system functionality to independent, loosely coupled, interoperable services. In this study we propose a new heterogeneous specification (Knapp *et al.*, 2010).

Kontogiannis *et al.* (2007) stated that service orientation is a very promising paradigm for large systems. service-orientation is not the solution to all problems. There are certain types of problems, such as interoperability, integration, choreography, context awareness, design of ultra large scale distributed service-based applications, where service-orientation can be of benefit. To trigger the proactive adaptation of a service-oriented system, pending failures need to be predicted. It is important that such a failure prediction is accurate, such as to avoid the execution of unnecessary proactive adaptations, as well as not to miss proactive adaptation opportunities (Metzger, 2011).

Application product lines: Cavalcante *et al.* (2012) stated that one of the main challenges to use the software product line approach for cloud applications is how to find the true development costs, involving all of the cost associated with maintaining the services and data placed on the cloud. Software product line will typically require more upfront design and development before the first product is produced than building a one off product. Research has shown the cost break even point in using an SPL is reached by the third product. SPL's rely on a single

design that covers the entire problem area for all of the products to be produced. This is difficult in new problem domains that are not well understood or are rapidly changing, Hunt and McGregor (2006).

According to Voelter and Groher (2007), variability of features often has widespread impact on multiple artifacts in multiple lifecycle stages, making it a predominant engineering challenge in software product line engineering. The mapping from problem to solution domain can be formally described and automated using model-to- model transformations. The problem domain model might not contain all the information necessary for the transformation to populate the solution domain model. For manually written code it is more challenging, since a piece of hand-written code may implement any number of requirements. As product-line applications become largely dominating, challenging problems such as dynamic variability and features interaction require special emphasis (Aoumeur *et al*., 2009). One of the biggest challenges in the configuration process is its scalabilit. The scalability issues arise when the number of features increases in Software product line (Mohabbati *et al*., 2011).

Da Silva *et al*. (2010) reported that the software product lines has proved very effective in building large-scale software. However, few works seek to adjust the approach of software product line to applications in the context of semantic web because applications in this context assume the use of semantic services and intelligent agents. Therefore, it is necessary that there are assets that provide adequate interoperability both semantic services and intelligent agents. On the other hand, there are not methods and techniques consolidated to suit the approach of software product line to semantic web applications. Product lines introduce extra complexity in software development but offer high returns. Is it possible to predict when product line investment pays in a specific domain and environment Knauber and Succi (2001).

Carbon *et al*. (2008) described that the product line organizations need to continuously invest into their product line infrastructure to minimize its degeneration and thus maximize its viability. Besides feedback on a strategic level to monitor changing customer requirements, technical feedback with respect to reusing product line components needs to be provided from application to family engineering. Even when product line infrastructure has been set up perfectly, its value decreases over time because the optimal scope moves due to changes in the domain and customer expectations.

According to Knauber and Succi, (2000), product lines introduce additional complexity. In a sense they go against the common adage of "divide and conquer." Planning andor developing of more than one product at a time have to be managed technically and organizationally. However, the rate of innovation of the technology and the intrinsic nature of software products do not let alternatives to developers: Users like to jump into the bandwagon of new products and old products often drive preferences to new products.

COTS integration: Megas *et al*. (2013) stated that the COTS integration introduce many problems related to safety, reliability and security, among others. These problems need to be addressed. Another issue not addressed has to do with maintainability of COTS-based systems. As there is no control over the evolution of the COTS products used, maintainability represents a great challenge because upgrades are frequently not compatible, products become obsolete and they end up unsupported by vendors, which can easily make the system unusable. COTS-based systems approach is often considered essential to building highly functional, competitive systems this approach also poses many challenges. Foremost among these is the identification of compatible components that meet the functional requirements of the system under development Seacord *et al*. (2002).

According to Yakimovich *et al*. (1999), the use of commercial-off-the-shelf (COTS) products creates a software integration problem. Whether a single COTS software component is being integrated into a software system, or the whole system is being built primarily from COTS products. This integration may require considerable effort and affect system quality. The use of COTS software product introduces new problems and risks for the organizations. For example, a COTS product is difficult to be integrated into a system; COTS product cannot provides capabilities vendor claims to have at acceptable level of quality; or the product is no longer supported because the vendor is out of a business. These, in turn, lead to slipped schedule, cost overrun and unreliable and unsatisfied system (Vantakavikran and Prompoon, 2007).

According to Warboys *et al*. (2005), the inherented contradiction between using long-lived, general-purpose COTS components and the demand for highly adaptable information systems creates a challenging problem. Classical software engineering has often given inadequate attention to the dynamic environment when developing COTS based systems. Ignoring this problem leads to legacy systems that are unaware of their changing environment. The encapsulation, heterogeneity and complexity of COTS components make the integration work challenging. Furthermore, due to inaccessibility of the source code, reuse of COTS software cannot leverage traditional code-reuse techniques and needs additional

glue code for adaptations. COTS components typically present interoperability problems and interface mismatches, which should be addressed at the architectural level (Ermagan *et al*., 2007).

Kumar *et al*. (2010) stated that the success of the solution depends on the selection of the appropriate COTS product lines and vendors that address the solution during deployment and post deployment support. There can be pitfalls in harmonization of COTS package with platform, operating system components and other COTS software packages. Architecture compatibility is another significant issue that needs to be addressed when considering COTS integration. Incompatible COTS design assumptions can cause serious interoperability problems affecting data, control, timing and service provision (Stavridou, 1997).

Program libraries: Narfelt and Schefstrom, (1985) studied that the increased automation of software production require that the automating tools have access to a lot of reliable input in a form suitable for mechanic manipulation. The problem of plain graphs is that they are too simple to be really useful here: describing software by just giving nodes and edges is not very helpful since we know too little about their meaning. The idea of a database as a central facility of a programming environment is explored, taking the Ada program library as a starting point. The database used is based on a node model Since compilation speed often seems to be a problem of Ada compilers, it is easy to reject the approach just because of this risk. Separate compilation issues must be considered at two stages of the compilation process, namely: During semantic analysis and at binding time. Before semantic analysis, at least the correct order of compilation has to be ensured.

Jarvi *et al*. (2004) studied that it is difficult to fully leverage the potential of generic programming in modern software construction. The main problem with the generic programming is that it fails to encapsulate associated types and their constraints into a single concept abstraction. Every use of a concept as a constraint of a generic function or a refinement declaration must list all of its associated types and all constraints on those types. In a concept with several associated types, this becomes burdensome. The lack of retroactive modeling is not an inherent problem of subtype-based constraints. Retroactive subtyping can be implemented for object-oriented languages. With parallel and distributed algorithms, there are additional challenges in developing a library standard in terms of concept taxonomies.

According to Wells *et al*., (1985), the problems of block structuring have been well documented. Nevertheless, we believe that this hierarchical name scoping concept is too "natural" to be readily discarded. Moreover, some of its problems are easily resolved by straightforward extension and/or incorporation of other independently useful language features. The four major problems of block structuring discussed in the literature are: Indiscriminate access, difficulty of separate compilation, large separation of a variable's declaration from its use and absence of nonlocal-variable information within blocks. These problems along with the need for a secure and practical data abstraction mechanism have motivated the introduction of "modules".

Program generators: According to Sampath *et al*. (2007), program generators are programs and at first sight, the traditional techniques for testing programs ought to be applicable to program generators as well. However, the rich semantic structure of the inputs and outputs of program generators poses unique challenges that have so far not been addressed sufficiently in the testing literature. A tester of a program generator is interested in 7generating a suite of test-models such that it has complete coverage over the problematic aspects of syntax and semantics of the modeling language. The value of a program generator is often tied so closely to a software domain that there is little general and reusable knowledge to transmit to other generator researchers Smaragdakis *et al*. (2004).

Markov *et al*. (2011) reported that the widespread use of data acquisition and control systems poses a serious challenge to software developers in view of diverse, severe and conflicting requirements. There will always be some code that needs to be hand-written. The actual amount varies from project to project. Usually, the generated code acts as support for this hand-written code or a library supporting the specific problem domain. For database code, the database must be well formed. Generators generally do not work well with databases that have special, "unique" design features.

Punyashthiti and Smith (1996) stated that a program generator can be divided into three parts, the dialogue module, an intermediate module and the code generation module. The dialogue module communicates with the user to get the specifications for the problem. The code generation module then uses the specifications to produce the program in the desired high-level language. Many of the problems besieging our industry today, including lack of good-quality personnel, lack of widely accepted standards problems with reputation and acceptance by the general public and the rapidly decreasing price of hardware with respect to the labor-intensive costs of software production, are problems that can be solved today and in the future by the program-generating tool (Roth, 1982).

Smaragdakis *et al.* (2004) studied the technical problems, metapro-gramming has a need for better language constructs, type systems and analyses to ensure safety. The challenge is to design a static checking mechanism that is expressive enough for common program generators. The term program generator is used here to denote a processor for a language more problem specific than the familiar, procedure-oriented, higher-level languages but less so than the nonprocedural problem-oriented languages which have been developed for rather narrow classes of problems Metzner (1977).

Aspect-oriented software development: Previous research in AOSD has focused primarily on the activities of software system design, problem analysis and language implementation. Although, testing is known to be a labor intensive process that accounts for half the total cost of software development, research on testing in AOSD, especially automated testing, has not been sufficiently conducted. Although, an aspect oriented design or implementation can lead to a better system architecture or a disciplined coding style. There are two major problems to be addressed when we measure data coverage for AspectJ programs. One is to identify which methods belong to aspects and the other is to characterize input-data values for these methods because input-data values can involve the states of receiver objects and arguments Xie and Zhao (2007).

Aspects have been defined in the implementation phase of software development, but lack clear understanding in the early phases of software development. Without this early focus on aspects, the benefits of aspect-oriented programming are lost. there is no complete and systematic methodology for AOSD within the early stages of the development lifecycle that focuses on identifying aspects from both functional and non-functional requirements Budwell and Mitropoulos (2008).

According to Brichau *et al.* (2006), as new software engineering techniques emerge, there's a cognitive shift in how developers approach a problem's analysis and how they design and implement its software-based solution. Future software engineers must be appropriately and effectively trained in new techniques' fundamentals and applications. advanced development paradigms have been developed that can be related to AOSD, either because they're complementary or because they target the same problems as AOSD. Design level metrics are currently not available for Aspect Oriented Software Development (AOSD). However, research in software measurement must adapt to the emergence of new software development paradigms. Metrics for these new design paradigms must be defined based on models that are suitable for them Babu and Vijayalakshmi (2008).

Rashid *et al.* (2010) stated the inversion of control provided by AOSD techniques helps separate crosscutting concerns. At the same time, however, the "new" aspectual modules must be tested. This is particularly problematic when trying to test aspects' effects on a program's exception flows. As aspects extend or replace existing functionality at specific joinpoints in the code execution, their behavior may raise new exceptions that can flow through the program execution in unexpected ways. When specifying point-cuts in Aspect-oriented software development, developers have in different situations different conceptual models in mind. Aspect-oriented programming languages are usually capable to support only a small subset of them, but not all. In order to communicate aspect oriented design among developers, though, it is inevitable that the underlying conceptual model used in its join point selections remains unchanged Stein *et al.* (2006).

According to Mehmood and Jawawi (2011), aspect-oriented programming languages have come to the mainstream of software development due to their distinctive features to provide better modularization and separation of concerns. Majority of approaches address structure diagrams only, a fact that limits them to partial code generation. There is a need for research that incorporates behavior diagrams, in order to achieve long term goal of full code generation from aspect-oriented models. Timeliness and criticality of a process are the two main concerns when designing real-time systems. In addition to that embedded systems are bounded by limited resources. There are several problems with designing schedulers for embedded and real-time systems like the need for more flexible policies which are not hard-coded in the system Cheng and Papadopoulos (2006).

Gerami and Ramsin (2011) stated that the detection and separation of crosscutting concerns during early phases of software development improves software evolvability and modifiability. However, due to the relative novelty of the Aspect-Oriented (AO) approach, researchers have concentrated more on AO programming and modeling rather than on AO methodologies and processes. Consequently, organizations still suffer from a lack of concrete methodology support for Aspect-Oriented Software Development. The AO modeling approaches are less mature and they are not fully supported by most modeling languages; AOSD thus faces serious challenges at the modeling level. Separation of concerns is one of the main tenets of software engineering-allowing developers to reason about software systems in sensible portions, regardless which phase of the lifecycle they are working in Baniassad *et al.* (2006).

Software systems and the concerns addressed by them are becoming increasingly complex hence posing

new challenges to the mainstream software engineering paradigms. The object-oriented paradigm is not sufficient to modularise crosscutting concerns, such as persistence, distribution, error handling and security, because they naturally crosscut the boundaries of other concerns. AOSD tackles the specific problem of managing crosscutting concerns throughout the software development lifecycle. Identifying aspects in required documents, is a challenging task which requires intuitive guidelines as well as tool support (Rashid *et al*., 2006).

Configurable vertical applications: Once vertical applications are identified, they require a set of a specific knowledge be placed in a toolset. The resulting toolset must have the ability to be robust and applicable to not only a specific problem, but a class of problems or a genre of engineering problems. Finite element modeling software has long been the face of the finite element solver. The user interacts with the solver through a graphical interface which aids them in creating the necessary data structures to simulate their engineering problems. The user interface's ability display the range of functions and guide the user through the process become problematic (Ouyang *et al*., 2009).

CONCLUSION

Despite the rapid advancement in information and communication technology over the last decade, there is a limit evidence suggesting the adaption of software reusability. In this study, we have presented a literature of the most up-to-date research work published on software reusability. This review study helps the information and communication technology industry to see clearly how software reusability can benefit them by adapting the software reusability approaches, the study has also attempted to identify the remaining challenges to be addressed and promising directions in the domain of software reusability for future research. We found some evidence that practicing software reusability brings few advantages for the IT industry. Therefore, we advocate the adaption of software reusability in the IT industry.

REFERENCES

Al-Bashayreh, M.G., N.L. Hashim and O.T. Khorma, 2012a. A survey on success factors to design application frameworks to develop mobile patient monitoring systems. Proceedings of the IEEE EMBS Conference on Biomedical Engineering and Sciences, December 17-19, 2012, Langkawi pp: 57-62.

Al-Bashayreh, M.G., N.L. Hashim and O.T. Khorma, 2012b. Towards successful design of context-aware application frameworks to develop mobile patient monitoring systems using wireless sensors. Proceedings of the IEEE Conference on Open Systems, October 21-24, 2012, Kuala Lumpur, pp: 1-6.

Al-Tahat, K.S., T.M.T. Sembok and S.Bin Idris, 2001. Using design patterns in the development of a planner-based courseware system. Proceedings of the IEEE Region 10th International Conference on Electrical and Electronic Technology, Volume 2, August 19-22, 2001, Singapore, pp: 873-876.

Aoumeur, N., K. Barkaoui and G. Saake, 2009. Validating and dynamically adapting and composing features in concurrent product-lines applications. Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, April 14-16, 2009, San Francisco, CA., USA., pp: 138-146.

Aoyama, M., 2000. Evolutionary patterns of design and design patterns. Proceedings of the IEEE International Symposium on Principles of Software Evolution, November 1-2, 2000, Kanazawa, Japan, pp: 110-116.

Aversano, L., G. Canfora, L. Cerulo, C. del Grosso and M. di Penta, 2007. An empirical study on the evolution of design patterns. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 3-7, 2007, Dubrovnik, Croatia, pp: 385-394.

Babu, C. and R. Vijayalakshmi, 2008. Metrics-based design selection tool for aspect oriented software development. ACM SIGSOFT Software Eng. Notes, Vol. 33 10.1145/1402521.1402522

Baniassad, E.L., K. Chen, S., Chiba, J. Hannemann, H. Masuhara, S. Ren and J. Zhao, 2006. 2nd Asian workshop on aspect-oriented software development (AOAsia). Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, September 18-22, 2006, Tokyo, Japan, pp: 375-375.

Basha, N.M.J. and S.A. Moiz, 2012. Component based software development: A state of art. Proceedings of the International Conference on Advances in Engineering, Science and Management, March 30-31, 2012, Nagapattinam, Tamil Nadu, pp: 599-604.

Bayley, I. and H. Zhu, 2010. Formal specification of the variants and behavioural features of design patterns. J. Syst. Software, 83: 209-221.

Bohn, S., W. Korb and O. Burgert, 2008. A process and criteria for the evaluation of software frameworks in the domain of computer assisted surgery. Med. Biol. Eng. Comput., 46: 1209-1217.

Brichau, J., R. Chitchyan, S. Clarke, A. Rashid and A. Garcia *et al*., 2006. A model curriculum for aspect-oriented software development. IEEE Software, 23: 53-61.

Budhija, N. and S.P. Ahuja, 2011. Review of software reusability. Proceedings of the 1st International Conference on Computer Science and Information Technology, January 2-4, 2011, Bangalore, India, pp: 113-115.

Budwell, C.C. and F.J. Mitropoulos, 2008. The SLAI methodology: An aspect-oriented requirement identification process. Proceedings of the International Conference on Computer Science and Software Engineering, Volume 2, December 12-14, 2008, Wuhan, Hubei, pp: 296-301.

Burgin, M., H.K. Lee and N. Debnath, 2004. Software technological roles, usability and reusability. Proceedings of the IEEE International Conference on Information Reuse and Integration, November 8-10, 2004, Las Vegas, Nevada, USA., pp: 210-214.

Carbon, R., J. Knodel, D. Muthig and G. Meier, 2008. Providing feedback from application to family engineering: The product line planning game at the testo AG. Proceeding of the 12th International Software Product Line Conference, September 8-12, 2008, Limerick, Ireland, pp: 180-189.

Carro, M., D. Karastoyanova, G.A. Lewis and A. Liu, 2011. Third international workshop on principles of engineering service-oriented systems (PESOS 2011). Proceedings of the 33rd International Conference on Software Engineering, May 21-28, 2011, Waikiki, Honolulu, HI., USA., pp: 1218-1219.

Cavalcante, E., A. Almeida, T. Batista, N. Cacho and F. Lopes *et al*., 2012. Exploiting software product lines to develop cloud computing applications. Proceedings of the 16th International Software Product Line Conference, Volume 2, September 2-7, 2012, Salvador, Brazil, pp: 179-187.

Cheng, J., 1994. A reusability-based software development environment. ACM SIGSOFT Software Eng. Notes, 19: 57-62.

Cheng, P.L. and G.A. Papadopoulos, 2006. A review of aspect-oriented software development techniques used in real-time and embedded systems software. Proceedings of the ITGA FA 6.2 Workshop on Model-Based Testing and GI/ITG Workshop on Non-Functional Properties of Embedded Systems, 13th GI/ITG Conference-Measuring, Modelling and Evaluation of Computer and Communication, March 27-29, 2006, Berlin, Germany, pp: 1-12.

Chenghao, G., W. Min and Z. Xiaoming, 2010. A wrapping approach and tool for migrating legacy components to web services. Proceedings of the 1st International Conference on Networking and Distributed Computing, October 21-24, 2010, Hangzhou, China, pp: 94-98.

Crnkovic, I., 2012. Managing complexity and predictability in embedded systems: Applying component-based development. Proceedings of the 2nd International Workshop on Software Engineering for Embedded Systems, June 9, 2012, Zurich, Switzerland, pp: 1-1.

Da Silva, A.P., E. Costa, I.I. Bittencourt, P.H.S. Brito, O. Holanda and J. Melo, 2010. Ontology-based software product line for building semantic web applications. Proceedings of the Workshop on Knowledge-Oriented Product Line Engineering, October 17-21, 2010, Reno, NV., USA., pp: 1.

De Oliveira, R.P., A.F. do Prado, W.L. de Souza and M. Biajiz, 2009. Development based on MDA, of ubiquitous applications domain product lines. Proceedings of the 8th IEEE/ACIS International Conference on Computer and Information Science, June 1-3, 2009, Shanghai, pp: 1005-1010.

Ermagan, V., C. Farcas, E. Farcas, I.H. Kruger and M. Menarini, 2007. A service-oriented blueprint for COTS integration: The hidden part of the iceberg. Proceedings of the 2nd International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques, May 20-26, 2007, Minneapolis, MN., USA., pp: 10.

Fant, J.S., H. Gomaa and R.G. Pettit, 2011. Architectural design patterns for flight software. Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, March 28-31, 2011, Newport Beach, CA., pp: 97-101.

Fazal-e-Aminm, A.K. Mahmood and A. Oxley, 2010. Proposal for evaluation of software reusability assessment approach employing a mixed method. ACM SIGSOFT Software Eng. Notes, 33: 1-4.

Fazal-e-Amin, A.K. Mahmood and A. Oxley, 2011. A mixed method study to identify factors affecting software reusability in reuse intensive development. Proceedings of the National Postgraduate Conference, September 19-20, 2011, Kuala Lumpur, pp: 1-6.

Gerami, M. and R. Ramsin, 2011. A framework for extending agile methodologies with aspect-oriented features. Proceedings of the IEEE 5th International Conference on Research Challenges in Information Science, May 19-21, 2011, Gosier, pp: 1-6.

Gill, N.S., 2006. Importance of software component characterization for better software reusability. ACM SIGSOFT Software Eng. Notes, 31: 1-3.

Gill, N.S. and P. Tomar, 2010. Modified development process of component-based software engineering. ACM SIGSOFT Software Eng. Notes, 35: 1-6.

Gill, P.E., W. Murray, S.M. Picken and M.H. Wright, 1979. The design and structure of a Fortran program library for optimization. ACM Trans. Math. Software, 5: 259-283.

Gupta, M., A. Pande and A.K. Tripathi, 2011. Design patterns detection using SOP expressions for graphs. ACM SIGSOFT Software Eng. Notes, 36: 1-5.

Hsueh, N.L., L.C. Wen, D.H. Ting, W. Chu, C.H. Chang and C.S. Koong, 2011. An approach for evaluating the effectiveness of design patterns in software evolution. Proceedings of the IEEE 35th Annual Computer Software and Applications Conference Workshops, July 18-22, 2011, Munich, Germany, pp: 315-320.

Hu, K., Z. Guo, Y. Jiang, Y. Feng and F. Shen, 2012. Component-based development framework for ocean information system. Proceedings of the MTS/IEEE Oceans, October 14-19, 2012, Hampton Roads, VA., pp: 1-7.

Hunt, J.M. and J.D. McGregor, 2006. Software product lines: A pedagogical application. J. Comput. Sci. Coll., 22: 295-302.

Jalender, B., A. Govardhan and P. Premchand, 2011. Breaking the boundaries for software component reuse technology. Int. J. Comput. Appl., 13: 37-41.

Jalender, B., A. Govardhan and P. Premchand, 2012. Designing code level reusable software components. Int. J. Software Eng. Appl., 3: 219-229.

Jarvi, J., A. Lumsdaine, D.P. Gregor, M. Kulkarni, D.R. Musser and S. Schupp, 2004. Generic programming and high-performance libraries. Proceedings of the 18th International Parallel and Distributed Processing Symposium, April 26-30, 2004, Santa Fe, New Mexico, USA., pp: 198-198.

Jensen, A.C. and B.H. Cheng, 2010. On the use of genetic programming for automated refactoring and the introduction of design patterns. Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, July 7-10, 2010, Portland, OR., USA., pp: 1341-1348.

Johnson, R.E., 1997. Components, frameworks, patterns. ACM SIGSOFT Software Eng. Notes, 22: 10-17.

Kahtan, H., N.A. Bakar and R. Nordin, 2012. Reviewing the challenges of security features in component based software development models. Proceedings of the IEEE Symposium on E-Learning, E-Management and E-Services, October 21-24, 2012, Kuala Lumpur, Malaysia, pp: 1-6.

Kamalraj, R., B.G. Geetha and G. Singaravel, 2009. Reducing efforts on software project management using software package reusability. Proceedings of the IEEE International Advance Computing Conference, March 6-7, 2009, Patiala, India, pp: 1624-1627.

Kaur, K., N. Mohan and P.S. Sandhu, 2012. Resuability of software components using J48 decision tree. Proceedings of the International Conference on Artificial Intelligence and Embedded Systems, July 15-16, 2012, Singapore, pp: 69-71.

Khazankin, R., D. Schall and S. Dustdar, 2011. Adaptive request prioritization in dynamic service-oriented systems. Proceedings of the IEEE International Conference on Services Computing, July 4-9, 2011, Washington, DC., USA., pp: 9-15.

Kim, Y. and E.A. Stohr, 1998. Software reuse: Survey and research directions. J. Manage. Inform. Syst., 14: 113-147.

Knapp, A., G. Marczynski, M. Wirsing and A. Zawlocki, 2010. A heterogeneous approach to service-oriented systems specification. Proceedings of the Symposium on Applied Computing, March 22-26, 2010, Sierre, Switzerland, pp: 2477-2484.

Knauber, P. and G. Succi, 2000. Software product lines: Economics, architectures and applications. Proceedings of the 22nd IEEE International Conference on Software Engineering, June 4-11, 2000, Limerick, Ireland, pp: 814-815.

Knauber, P. and G. Succi, 2001. Second ICSE workshop on software product lines: Economics, architectures and applications. Proceedings of the 23rd International Conference on Software Engineering, May 12-19, 2001, Toronto, Ontario, Canada, pp: 753-754.

Kontogiannis, K., G.A. Lewis, D.B. Smith, M. Litoiu, H. Muller, S. Schuster and E. Stroulia, 2007. The landscape of service-oriented systems: A research perspective. Proceedings of the International Workshop on Systems Development in SOA Environments, May 20-26, 2007, Minneapolis, MN., USA., pp: 1.

Kumar, P.A., S. Narayanan and V.M. Siddaiah, 2010. COTS integrations: Effort estimation best practices. Proceedings of the IEEE 34th Annual Computer Software and Applications Conference Workshops, July 19-23, 2010, Seoul, South Korea, pp: 81-86.

Kume, I., M. Nakamura and E. Shibayama, 2012. Toward comprehension of side effects in framework applications as feature interactions. Proceedings of the 19th Asia-Pacific Software Engineering Conference, Volume 1, December 4-7, 2012, Hong Kong, China, pp: 713-716.

Lee, M.S., S.G. Shin and Y.J. Yang, 2001. The design and implementation of Enterprise JavaBean (EJB) wrapper for legacy system. Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Volume 3, October 7-10, 2001, Tucson, AZ., pp: 1988-1992.

Lemley, M.A. and D.W. O'Brien, 1996. Encouraging software reuse. Stan. L. Rev., 49: 255-304.

Lewis, G., D. Smith, A. Metzger, A. Zisman and M. Pistore, 2010a. Report of the 2nd international workshop on principles of engineering service-oriented systems. ACM SIGSOFT Software Eng. Notes, 35: 30-33.

Lewis, G.A., A. Metzger, M. Pistore, D. Smith and A. Zisman, 2010b. 2010 ICSE 2nd international workshop on principles of engineering service-oriented systems (PESOS 2010). Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Volume 2, May 1-8, 2010, Cape Town, South Africa, pp: 429-430.

Li, X., 2010. A multi-agent based legacy information system integration strategy. Proceedings of the 2nd International Conference on Networking and Digital Society, Volume 2, May 30-31, 2010, Wenzhou, pp: 72-75.

Li, X.Y. and Y. Qian, 2009. A web service based enterprise information integration model. Proceedings of the 4th International Conference on Computer Science and Education, July 25-28, 2009, Nanning, pp: 1251-1254.

Ling, H., X. Zhou and Y. Zheng, 2010. Refactoring from object-oriented systems to service-oriented systems: A categorical approach. Proceedings of the International Conference Service Sciences, May 13-14, 2010, Hangzhou, China, pp: 214-218.

Lingyun, F., S. Guang and C. Jianli, 2010. An approach for component-based software development. Proceedings of the International Forum on Information Technology and Applications, Volume 1, July 16-18, 2010, Kunming, pp: 22-25.

Loo, K.N. and S.P. Lee, 2010. Representing design pattern interaction roles and variants. Proceedings of the 2nd International Conference on Computer Engineering and Technology, Volume 6, April 16-18, 2010, Chengdu, China, pp: V6-470-V6-474.

Mailloux, M., 2010. Application frameworks: How they become your enemy. Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, October 17-21, 2010, Reno, NV., USA., pp: 115-122.

Markov, E., I.E. Ivanov and V. Gueorguiev, 2011. Program generator architecture. Proceedings of the IEEE Meeting of Developments in E-Systems Engineering, December 6-8, 2011, Dubai, United Arab Emirates, pp: 564-569.

Marshall, J.J. and R.R. Downs, 2008. Reuse readiness levels as a measure of software reusability. Proceedings of the IEEE Geoscience and Remote Sensing Symposium, Volume 3, July 7-11, 2008, Boston, MA., pp: III-1414-III-1417.

McCarey, F., M.O. Cinneide and N. Kushmerick, 2008. Knowledge reuse for software reuse. Web Intell. Agent Syst., 6: 59-81.

Megas, K., G. Belli, W.B. Frakes, J. Urbano and R. Anguswamy, 2013. A study of COTS integration projects: Product characteristics, organization and life cycle models. Proceedings of the 28th Annual ACM Symposium on Applied Computing, March 18-22, 2013, Coimbra, Portugal, pp: 1025-1030.

Mehmood, A. and D.N.A. Jawawi, 2011. A comparative survey of aspect-oriented code generation approaches. Proceedings of the IEEE 5th Malaysian Conference in Software Engineering, December 13-14, 2011, Johor Bahru, pp: 147-152.

Metzger, A., 2011. Towards accurate failure prediction for the proactive adaptation of service-oriented systems. Proceedings of the 8th Workshop on Assurances for Self-Adaptive Systems, September 5-9, 2011, Szeged, Hungary, pp: 18-23.

Metzner, J.R., 1977. Program generator systems. Proceedings of the 9th Conference on Winter Simulation, Volume 2, December 5-7, 1977, Gaitersburg, MD., USA., pp: 694-700.

Mohabbati, B., M. Hatala, D. Gasevic, M. Asadi and M. Boskovic, 2011. Development and configuration of service-oriented systems families. Proceedings of the ACM Symposium on Applied Computing, March 21-24, 2011, Taichung, Taiwan, pp: 1606-1613.

Morisio, M., C.B. Seaman, A.T. Parra, V.R. Basili, S.E. Kraft and S.E. Condon, 2000. Investigating and improving a COTS-based software development. Proceedings of the 22nd International Conference on Software Engineering, June 4-11, 2000, Limerick, Ireland, pp: 32-41.

Mott, M.A. and F. Khan, 2000. Integration of commercial-off-the-shelf (COTS) video stimulus instrument with fielded Automated Test System (ATS). Proceedings of the IEEE Systems Readiness Technology Conference: Future Sustainment for Military and Aerospace, September 18-21, 2000, Anaheim, CA., USA., pp: 539-543.

Narfelt, K.H. and D. Schefstrom, 1985. Extending the scope of the program library. ACM SIGAda Ada Lett., 5: 25-40.

Newman, P. and G. Kotonya, 2011. A runtime resource-management framework for embedded service-oriented systems. Proceedings of the 9th Working Conference on Software Architecture, June 20-24, 2011, Boulder, CO., USA., pp: 123-126.

Nitto, D., 2009. Principles of engineering service oriented systems. Proceedings of the 31st International Conference on Software Engineering-Companion Volume, May 16-24, 2009, Vancouver, Canada, pp: 461-462.

Ouyang, H., T. Palmer and Q. He, 2009. A next generation software platform for LS-DYNA modeling and configurable vertical application development. Proceedings of the 7th European LS-DYNA Conference on Engineering Technology Association, May 14-15, 2009, Austria.

Palma, F., H. Farzin, Y. Gueheneuc and N. Moha, 2012. Recommendation system for design patterns in software development: An DPR overview. Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering, June 4, 2012, Zurich, pp: 1-5.

Pande, J., C.J. Garcia and D. Pant, 2013. Optimal component selection for component based software development using pliability metric. ACM SIGSOFT Software Eng. Notes, 38: 1-6.

Panwar, D. and P. Tomar, 2011. New method to find the maximum number of faults by analyzing reliability and reusability in component-based software. Proceedings of the 3rd International Conference on Trendz in Information Sciences and Computing, December 8-9, 2011, Chennai, India, pp: 164-168.

Parhomenko, A., I. Pavlyuchenko, A. Votinov and V. Kilmenko, 2003. Investigation and development of program libraries for KOMPAS CAD. Proceedings of the 7th International Conference of the Experience of Designing and Application of CAD Systems in Microelectronics, February 18-22, 2003, Lviv-Slavske, Ukraine, pp: 290-292.

Parsa, S. and L. Ghods, 2008. A new approach to wrap legacy programs into web services. Proceedings of the 11th International Conference on Computer and Information Technology, December 24-27, 2008, Khulna, pp: 442-447.

Punyashthiti, A. and D.L. Smith, 1996. Program generator for RS-232C instruments. Proceedings of the ACM symposium on Applied Computing, February 17-19, 1996, Philadelphia, PA., USA., pp: 487-492.

Ramirez, A.J. and B.H. Cheng, 2010. Design patterns for developing dynamically adaptive systems. Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, May 1-8, 2010, Cape Town, South Africa, pp: 49-58.

Rashid, A., A. Garcia and A. Moreira, 2006. Aspect-oriented software development beyond programming. Proceedings of the 28th International Conference on Software Engineering, May 20-28, 2006, Shanghai, China, pp: 1061-1062.

Rashid, A., T. Cottenier, P. Greenwood, R. Chitchyan and R. Meunier *et al*., 2010. Aspect-oriented software development in practice: Tales from aosd-europe. Computer, 43: 19-26.

Raza, M.S., S.H. Majoka and A. Mohsin, 2010. An integrated approach for developing semantic-mismatch free commercial off the shelf (COTS) components. Proceedings of the 8th International Conference on Frontiers of Information Technology, December 21-23, 2010, Islamabad, Pakistan, pp: 8.

Rittammanart, N., W. Wongyued and M.N. Dailey, 2008. ERP application development frameworks: Case study and evaluation. Proceedings of the 5th International Conference on Electrical Engineering/electronics, Computer, Telecommunications and Information Technology, Volume 1, May 14-17, 2008, Krabi, Thailand, pp: 173-176.

Roth, R.L., 1982. Program generators and their effect on programmer productivity. Proceedings of the National Computer Conference, June 7-10, 1982, Houston, Texas, pp: 351-358.

Rothenberger, M.A., K.J. Dooley, U.R. Kulkarni, and N. Nada, 2003. Strategies for software reuse: A principal component analysis of reuse practices. IEEE Trans. Software Eng., 29: 825-837.

Sagar, S., N.W. Nerurkar and A. Sharma, 2010. A soft computing based approach to estimate reusability of software components. ACM SIGSOFT Software Eng. Notes, 35: 1-4.

Sampath, P., A.C. Rajeev, K.C. Shashidhar and S. Ramesh, 2007. How to test program generators? A case study using flex. Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods, September 10-14, 2007, London, pp: 80-92.

Sanchez, P., D. Garcia-Saiz and M. Zorrilla, 2012. Software product line engineering for e-learning applications: A case study. Proceedings of the International Symposium on Computers in Education, October 29-31, 2012, Andorra la Vella, Andorra, pp: 1-6.

Sandhu, P.S. and H. Singh, 2008. Software reusability model for procedure based domain-specific software components. Int. J. Software Eng. Knowl. Eng., 18: 1063-1081.

Sandhu, P.S., Aashima, P. Kakkar and S. Sharma, 2010. A survey on software reusability. Proceedings of the 2nd International Conference on Mechanical and Electrical Technology, September 10-12, 2010, Singapore, pp: 769-773.

Seacord, R.C., G.A. Lewis and R. Bunting, 2002. COTS integration and evaluation: Introduction. Proceedings of the 10th International Workshop on Software Technology and Engineering Practice, October 6-8, 2002, Montreal, Canada, pp: 163-163.

Sharma, A., P.S. Grover and R. Kumar, 2009. Reusability assessment for software components. ACM SIGSOFT Software Eng. Notes, 34: 1-6.

Sharp, J.H. and S.D. Ryan, 2010. A theoretical framework of component-based software development phases. ACM SIGMIS Database, 41: 56-75.

Simanta, S., E. Morris, G.A. Lewis and D.B. Smith, 2010. Engineering lessons for systems of systems learned from service-oriented systems. Proceedings of the 4th Annual IEEE Systems Conference, April 5-8, 2010, San Diego, CA., pp: 634-639.

Singaravel, G., V. Palanisamy and A. Krishnan, 2010. Overview analysis of reusability metrics in software development for risk reduction. Proceedings of the International Conference on Computing Technologies, February 12-13, 2010, Tamil Nadu, India, pp: 1-5.

Singh, S., S. Singh and G. Singh, 2010. Reusability of the software. Int. J. Comput. Appl., 7: 38-41.

Singh, Y., P.K. Bhatia and O. Sangwan, 2011. Software reusability assessment using soft computing techniques. ACM SIGSOFT Software Eng. Notes, 36: 1-7.

Smaragdakis, Y., S.S. Huang and D. Zook, 2004. Program generators and the tools to make them. Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, August 24-25, 2004, Verona, Italy, pp: 92-100.

Sommerville, I., 2004. Software Engineering. 7th Edn., Addison-Wesley, New York, ISBN: 0321210263.

Souder, T. and S. Mancoridis, 1999. A tool for securely integrating legacy systems into a distributed environment. Proceedings of the 6th Working Conference on Reverse Engineering, October 6-8, 1999, Atlanta, GA., pp: 47-55.

Stavridou, V., 1997. COTS, integration and critical systems. Proceedings of the IEE Colloquium on Cots and Safety Critical Systems, January 28, 1997, London, UK., pp: 1-5.

Stein, D., S., Hanenberg and R. Unland, 2006. Expressing different conceptual models of join point selections in aspect-oriented design. Proceedings of the 5th International Conference on Aspect-Oriented Software Development, March 20-24, 2006, Bonn, Germany, pp: 15-26.

Stepan, P., 2011. Design pattern solutions as explicit entities in component-based software development. Proceedings of the 16th International Workshop on Component-Oriented Programming, June 20-24, 2011, Boulder, CO, USA., pp: 9-16.

Thung, P.L., C.J. Ng, S.J. Thung and S. Sulaiman, 2010. Improving a web application using design patterns: A case study. Proceedings of the International Symposium on Information Technology, Volume 1, June 15-17, 2010, Kuala Lumpur, pp: 1-6.

Van der Burg, S. and E. Dolstra, 2011. A self-adaptive deployment framework for service-oriented systems. Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 21-28, 2011, Waikiki, Honolulu, HI., USA., pp: 208-217.

Vantakavikran, P. and N. Prompoon, 2007. Constructing a process model for decision analysis and resolution on COTS selection issue of capability maturity model integration. roceedings of the 6th IEEE/ACIS International Conference on Computer and Information Science, July 11-13, 2007, Melbourne, Australia, pp: 182-187.

Vazquez-Salceda, J., W. Vasconcelos, J. Padget, F. Dignum and S. Clarke *et al*., 2010. ALIVE: A model-driven framework to develop dynamic, flexible, distributed service-oriented systems. Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, September 23-26, 2010, Timisoara, Romania, pp: 485-492.

Voelter, M. and I. Groher, 2007. Product line implementation using aspect-oriented and model-driven software development. Proceedings of the 11th International Software Product Line Conference, September 10-14, 2007, Kyoto, Japan, pp: 233-242.

Vuksanovic, I.P. and B. Sudarevic, 2011. Use of web application frameworks in the development of small applications. Proceedings of the 34th International Convention on MIPRO, May 23-27, 2011, Opatija, Croatia, pp: 458-462.

Warboys, B., B. Snowdon, R.M. Greenwood, W. Seet and L. Robertson *et al*., 2005. An active-architecture approach to COTS integration. IEEE Software, 22: 20-27.

Washizaki, H., H. Yamamoto and Y. Fukazawa, 2003. A metrics suite for measuring reusability of software components. Proceedings of the 9th International Symposium on Software Metrics, September 3-5, 2003, Sydney, Australia, pp: 211-225.

Wautelet, Y., S. Kiv, V. Tran and M. Kolp, 2010. Round tripping in component based software development. Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Volume 2, August 31-September 3, 2010, Toronto, Canada, pp: 261-264.

Weijiang, Q., Z. Weimei and S. Yongfeng, 2012. Design patterns applied in power system analysis software package. Proceedings of the International Conference on Industrial Control and Electronics Engineering, August 23-25, 2012, Xi'an, China, pp: 836-840.

Wells, M.B., M.A. Hug and R. Silver, 1985. Libraries as programs preserved within compiler continuations. ACM SIGPLAN Notices, 20: 83-91.

Xie, T. and J. Zhao, 2007. Perspectives on automated testing of aspect-oriented programs. Proceedings of the 3rd Workshop on Testing Aspect-Oriented Programs, March 12-13, 2007, Vancouver, Canada, pp: 7-12.

Yakimovich, D., J.M. Bieman and V.R. Basili, 1999. Software architecture classification for estimating the cost of COTS integration. Proceedings of the International Conference on Software Engineering, May 22, 1999, Los Angeles, CA., USA., pp: 296-302.

Zhang, B., L. Bao, R. Zhou, S. Hu and P. Chen, 2008. A black-box strategy to migrate GUI-based legacy systems to web services. Prcoeedings of the International Symposium on Service-Oriented System Engineering, December 18-19, 2008, Jhongli, Taiwan, pp: 25-31.

Zhang, H.Y., L. Zhang, C. Urtado, S. Vauttier and M. Huchard, 2012. A three-level component model in component based software development. Proceedings of the 11th International Conference on Generative Programming and Component Engineering, Volume 48, September 26-27, 2012, Dresden, pp: 70-79.

Zhou, J., D. Zhao and J. Liu, 2011. A lightweight component-based development approach for enterprise applications. Proceedings of the IEEE 35th Annual Computer Software and Applications Conference Workshops, July 18-22, 2011, Munich, Germany, pp: 335-340.