SOFTWARE REUSABILITY: LEVEL OF REUSE AND BENEFIT

Zulkar Nine, Faruqe Hasan, Sajedul Islam, Piyas Hossain & Rasel Hossain
Department of Computer Science and Engineering, City University, Dhaka 1215
Email: ∗zulkarnine43@gmail.com

ABSTRACT_ Software reuse is the use of software resources from all stages of the software development process in new applications. Given the high cost and difficulty of developing high-quality software, the idea of capitalizing on previous software investments is appealing. The biggest advantage of the building reusable software components is that it reduces the time and energy in developing any software. This paper surveys recent software-reuse research using a framework that helps identify and organize the many factors that must be considered to achieve the benefits of software reuse in practice and describes how to build the code level reusable components and how to design code level components. Finally providing coding guidelines, standards and best practices used for creating reusable code level components and guidelines and best practices for making configurable and easy to use. We argue that software reuse needs to be viewed in the context of a total systems approach that addresses a broad range of technical, economic, managerial, organizational, and legal issues and conclude with a summary of the major research issues in each of these areas.

KEYWORDS_

Software reuse; Reuse metrics; business and finance of reuse; Reuse Benefits and Costs; code Level; Reusable Component; reuse process, reuse technologies.

INTRODUCTION_

A reusable software part is Reusable software components can be simple like familiar push buttons, text fields list boxes, scrollbars, dialogs every thing visible in Java interface are reusable components. Software reuse is the use of engineering knowledge or artifacts from existing software components to build a new system [1][2]. There are many work products that can be reused, for example source code, designs, specifications, architectures and documentation [2].

In software engineering reuse is an important area where we can improve the productivity and quality of software [3]. Software reuse is the use of existing software or software knowledge to construct newsoftware [4]. A component is an object in thegraphical representation of application and that can interact with user. Reusable software components are designed to apply the power and benefit of reusable, interchangeable parts from other industries to the field of software construction [5].

The best example in this case is the manufacturers of Honda, Toyota and Suzuki cars would have not been so successful if these companies have not provided spare parts of their cars? Software companies have used the same concept to develop software in parts [6][7].

In brief the first way involves the technique for writing reusable software components and identifying those components, the second way involves the covering of the steps required for extending reusable software components; finally, the last way addresses testing and deploying your extensions and wrappers for reusable software components [8][9].

WHY REUSE SOFTWARE?

A good software reuse process facilitates the increase of productivity, quality, and reliability, and the decrease of costs and implementation time. An initial investment is required to start a software reuse process, but that investment pays for itself in a few reuses.

In short, the development of a reuse process and repository produces a base of knowledge that improves in quality after every reuse, minimizing the amount of development work required for future projects and ultimately reducing the risk of new projects that are based on repository knowledge.

BACKGROUND AND FEASIBILITY STUDY_

Software reuse is generally defined as the use of previously developed software resources from all phases of the software life cycle in new applications by various users such as programmers and systems analysts [10].

Questions related to the definition and scope of software reuse have been examined by a number of researchers [11, 12]. As in any new field, the terminology of reuse is evolving. As discussed, the tenninology becomes less standard when the details of reuse techniques and metrics for measuring the costs and benefits of reuse are considered.

PROPOSED MODEL

We will discuss this paper about level of reuse and benefit of reuse. we have shown business and finance of reuse, code levels of reuse, approaches supporting software reuse, cost-benefit models, benefits achieved through software reuse, code level components reuse, reuse capability maturity models, modifying and integrating reusable resources, summary & advanatges of reuse which is very helpful when anybody reuse.

BUSINESS AND FINANCE OF REUSE_

The ultimate purpose of software engineering and systematic software reuse is to improve the quality of the products and services that a company provides and, thereby, maximize profits. It is easy to lose sight of this goal when considering the technical challenges of software reuse and yet, software reuse will only succeed if it makes good business sense. Capital can be expended by an organization in many ways to maximize return to shareholders. Software reuse will only be chosen if a good case can be made that it is the best alternative choice for use of capital. More recent work has extended the return on investment analysis to include benefits from strategic market position [13].

APPROACHES SUPPORTING SOFTWARE REUSE_

- Application Frameworks
- Application product lines
- Aspect-oriented software development
- Component-based Development
- Configurable vertical applications
- COTS (Commercial-O-The-Shelf) integration
- Design Patterns
- Legacy system wrapping
- Program generators
- Program libraries
- Service-oriented systems

• **Application Frameworks:** Collections of concrete and abstract classes that can be adapted and extended to create application systems. It is used to implement the standard structure of a for a specific development environment. A framework is an incomplete implementation plus conceptually complete design. Application frameworks became popular with the rise of, since these tended to promote a standard structure for applications.

• **Application Product Lines:** Application product lines, or Application development, refers to methods, tools and techniques for creating a collection of similar product line systems from a shared set of software assets using a common. An application type is generalized around a common architecture so that it can be adapted in different ways for different customers. A type of application system reuse. Adaptation may involve component and system configuration; selecting from a library of existing components; adding new components to the system; or modifying components to meet new requirements [14].

• **Aspect-Oriented Software Development:** Aspect-oriented software development (AOSD) is an emerging software development technology that seeks new modularizations of software systems in order to isolate secondary or supporting functions from the main program's business logic. AOSD allows multiple concerns to be expressed separately and automatically unified into working systems [15].

• **Component-Based Development:** Systems are developed by integrating components (collections of classes) that conform to component-model standards. By adopting a component-based development approach you will have the option of buying off-the-shelf components from third parties rather than developing the same functionality inhouse [16].

• **Configurable Vertical Applications:** Configurable vertical application is a generic system that is designed so that it can be configured to the needs of specific system customers [17]. An example of a vertical application is software that helps doctors manage patient records, insurance billing, etc

• **COTS Integration:** By integrating existing application systems System is developed. A type of application system reuse. A commercial off–the-shelf (COTS) item is one that is sold, leased, or licensed to the general public. [18]

• **Design Patterns:** A design pattern is a recurring solution for repeatable problem in software design. Design Pattern is a template for how to solve a problem that can be used in many different situations [19].

• **Legacy System Wrapping:** By wrapping a set of defining interfaces by legacy systems provides access to interfaces. By rewriting a legacy system from scratch can create an equivalent functionality

information system based on modern software techniques and hardware [20].

• **Program Generators:** Program Generator is a program that enables an individual to easily create a program of their own with less effort and programming knowledge. With a program generator a user may only be required to specify the steps or rules required for his or her program and not need to write any code or very little code A generator system embeds knowledge of a particular type of application and can generate systems or system fragments in that domain. Program Generators Involves the reuse of standard patterns and algorithms [21].

• **Program Libraries:** Function and class libraries implementing commonly used abstractions are available for reuse. Libraries contain data and code that provides necessary services to independent programs. This idea encourages the exchanging and sharing of data and code.

• **Service-Oriented Systems:** SOA is a set of methodologies and principles for developing and designing software in the form of component. These components are developed by linking shared services that may be externally provided. An enterprise system often has applications and a stack of infrastructure including databases, operating systems, and networks [22].

LEVELS OF REUSE_

Reuse is divided into following four levels

1. Code level components (modules, procedures, subroutines, libraries, etc.)

2. Entire applications

3. Analysis level products

4. Design level products

The most frequently used component reuse is code level. Examples for code level component reuse are standard libraries and popular language extensions are the most obvious examples. In this case the level of abstraction is low for these components and the expected amount of reuse is low. For many real world problem domain reusing entire applications with little or no modification will give a high reuse when compared to code level component reuse. Using entire application means using commercial-off-the-shelf packages (COTS) or minimal adaptation of a specialized product applied to a new customer [23].

COST-BENEFIT MODELS AND REUSE METRICS

The central idea of reuse cost-benefit models is to develop mathematical relationships that express the benefits and costs of reuse in terms of metrics that can be captured in a software development organization. A typical model was developed by Gaffney and Durek [24] and is briefly described here. Let:

C = cost of software development with reuse for a given product relative to the cost of the same product if it were built with all new code;

R = reuse rate or proportion of reused code in the product;

b = cost, relative to that for developing new code, of incorporating reused code into the new product;

E = the relative cost of creating reusable code compared with normal code and

N = the number of the number of uses over which the reusable code cost is to be amortized.

This approach has been used to develop a multiperiod retum-on-investment model to determine if the initial investment in a software reuse program is worthwhile [25]. The basic idea is to convert to dollar terms by multiplying each term by estimates of the software size (in LOC) and development cost per LOC, and to add a term that estimates the maintenance benefits from the reduced error rates obtainable with reuse. A different class of model is needed to develop cost estimates for individual projects.

One approach is to modify the well-known COCOMO software cost estimation model [26] to allow for different types of reuse, the differential costs of developing for and with reuse, and the stage of the development life cycle in which reuse occurs While the above discussion has focused on metrics based on lines of code, it is not clear if this is a suitable basis for measurement in CASE or object-oriented environments. For example, Banker et al. describe metrics for evaluating the effectiveness of CASE tool reuse repositories in terms of the software objects produced and reused rather than in terms of LOC or FP.

QUANTITATIVE BENEFITS ACHIEVED THROUGH SOFTWARE REUSE_

• **Nippon Electric Company** Achieved 6.7 times higher productivity and 2.8 times better quality through 17% reuse. Nippon Electric company improved software quality 5-10 times over a seven-year period through the use of unmodified reuse components and achieved a better quality in the domain of basic system software development and in the domain of communication switching systems.

• **Toshiba** saw a 20-30% reduction in defects per line of code with reuse levels of 60%

• **Raytheon** achieved a 50% productivity increase in the MIS domain from 60% reuse using COBOL

• **Japanese industry study** 15-50% increase in productivity; 20-35% reduction in customer complaints; 20% reduction in training costs;10-50% reduction in time to produce the system

• **Simulator system developed for the US Navy** Increase of approximately 200% in number of SLOC produced per hour

• **NASA Report** Reduction of 75% in overall development effort and cost

• **AT&T** reported a 50% decrease in time-to-market for 40-90% reuse

• **Raytheon Missile Systems** experienced 1.5 times increase in productivity from 4060% reuse • SofTech had a 10-to-20 times increase in productivity for reuse greater than 75%

CODE LEVEL COMPONENT REUSE_

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 component is evaluated across a number of topic levels, each of the level which provides guidance about what one can expect at each reuse level. The topic levels currently defined are:

Level 1: Documentation

Level 2: Extensibility

Level 3: Intellectual Property Issues

Level 4: Modularity

Level 5: Packaging

Level 6: Portability

Level 7: Standards compliance

Level 8: Support

Level 9: Verification and Testing

Level	Summary
Level 1	Limited reusability; the software is not recommended for reuse.
Level 2	Initial reusability; software reuse is not practical.
Level 3	Basic reusability; the software might be reusable by skilled users at substantial effort, cost and risk.
Level 4	Reuse is possible; the software might be reused by most users with some effort, cost, and risk
Level 5	Reuse is practical; the software could be reused by most users with reasonable cost and risk.
Level 6	Software is reusable; the software can be reused by most users, although there may be some cost and risk.
Level 7	Software is highly reusable; the software can be reused by most users with minimum cost and risk.
Level 8	Demonstrated local reusability; the software has been reused by multiple users.
Level 9	Proven extensive reusability; the software is being reused by many classes of users over a wide range of systems.

HOW TO BUILD CODE LEVEL REUSABLE COMPONENTS_

A code level reusable software component is self-contained and has clearly defined boundaries with respect to what it does and does not do.For those users who need to modify the internals/functionality of the component in some way, for example to add a feature, or fix a previously undiscovered defect, a clear, unambiguous, and understandable specification for the component will be required.This allows users to modify implementation details, assuming source code is available and to build code level reusable components [27].

Finally, it is critical that the component is correctly licensed and full details are made available to the end user [28]

The following ways to build code level reusable components

• Class libraries

• Function libraries

• Design patterns

• Framework Classes

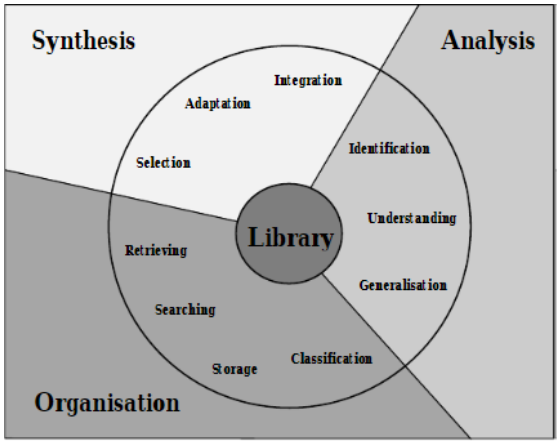
Class libraries

Class libraries are the object-oriented version of function libraries. Classes provide better abstraction mechanisms, better ability and adaptability than functions do. Reusability has greatly from concepts like inheritance, polymorphism and dynamic binding. In many class libraries there are classes devoted to generic data structures like lists, trees and queues. The major problem with class libraries is

that they consist of families of related components. Thus members of families have incompatible interfaces. Often several families implement the same basic abstraction but have interfaces. This makes libraries hard to use and makes interchanging components. Also, most class libraries are not scalable [29].

Function libraries

Functions are the most common form of reusable components. For many programming languages, standard libraries have been, for example, for input/output or mathematical functions. A few decades ago, languages had much functionality in the language itself. Later on, the trend was towards lean languages with standard libraries for various functionalities. There are many examples of function libraries, from collections of standard routines (e.g., the C standard libraries) to domain libraries (e.g., for statistics or numerical purposes) [30].



Design patterns

To save time and effort, it would be ideal if there was a repository which captured such common problem domains and proven solutions. In the simplest term, such a common solution is a design pattern. Design patterns can improve the structure of software, simplify maintenance, and help avoid architectural drift. This leads you to find a new solution for the similar problem each time.

A subsystem is a set of classes with high cohesion among themselves and low coupling to classes outside the subsystem [31. Example design patterns are Model/View/Controller (MVC), Blackboard, Client/Server, and Process Control. Design patterns can correspond to subsystems, but often they have level of granularity. Design patterns have been to avoid dependence on classes when creating objects, on particular operations, representation or implementation, on particular algorithms, and on inheritance as the extension mechanism [32].

To this end, an application is divided into three core components: the model, the view, and the controller. Each of these components handles a different set of tasks. The architecture of MVC shown in below figure.

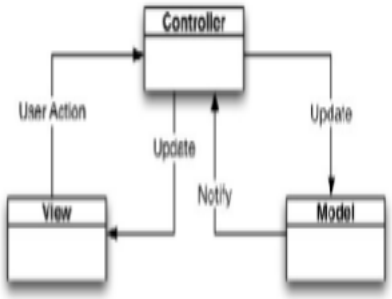


Figure 1.MVC architecture

Framework Classes

Frameworks are flexible collections of abstract and concrete classes designed to be extended and for reuse. Components of class libraries can serve as discrete, stand-alone, context-independent parts of a solution to a large range of applications, e.g., collection classes. Reuse of framework classes inherit the overall design of an application made by experienced software engineers and can concentrate on the application's functionality [33].

The major advantage of framework classes over library classes is that frameworks are concerned with conventions of communication between the components [34]. Today the combination of components from class libraries is the exception rather than the rule. This is because there is some implicit understanding of how components work together.

Framework is set of reusable software program that forms the basis for an application. Building Reusable Frameworks help the developers to build the application quickly. These are useful when reusing more than just code level component. Frameworks are having well written class libraries. By reusing these class libraries. we will build the code level reusable software components.

REUSE DEVELOPMENT KNOWLEDGE_

Level	Supplier-Related Knowledge	Customer-Related Knowledge
Environmental	**Technology transfer knowledge:** consists of knowledge about such things as the organizational impact of software technology, personnel training, computer literacy, and so forth.	**Utilization knowledge:** describesthe business context in which the software product will be used.
External	**Development knowledge:** deals with the planning and management of software projects such as cost and schedule estimation, test plans, benchmarking, and others.	**Application-area knowledge:** deals with th underlying models for the application to do main

REUSE CAPABILITY MATURITY MODELS

As described above, reuse practices in companies vary from ad-hoc, occasional reuse by individual programmers to planned and carefully executed programs of reuse involving new organizational entities and investments in the development of reusable resources and a supporting environment. It is important for organizations to understand their progress in the area of reuse relative to other companies and industry best practice. In the software area, the "Capability Maturity Model (CMM)" developed by SEI [35] describes the evolution of an organization's development processes from a chaotic ad-hoc state to a state of maturity in which industry-wide best practices are the norm. Similar maturity assessment models have been developed for reuse: Some models are add-ons to the SEI CMM, while others have reuse as the major theme [36]. To date, there has been little experience of CMM in the reuse area. However, if such models can be validated, practitioners and researchers will have a valuable tool to determine the overall state of reuse in individual organizations and in the software industry as a whole.

MODIFYING AND INTEGRATING REUSABLE RESOURCES

After reusable resources are retrieved and understood, they must be modified and integrated into the target system. Unfortunately, the complex process of integrating reusable resources into the target system is usually left entirely to the software developer. Current technologies for modifying reusable resources focus mainly on parameterized code resources. However, even highly parameterized software modules are shaped by prior implementation decisions and can be difficult to modify for use in a new context. Object-oriented languages such as C++ attack this problem by using message passing and inheritance as integrating principles.

The UNIX pipe mechanism provides a limited form of integration in which one program's outputs are connected to another program's inputs to construct more complex programs [37]. Another promising approach is adopted in the PARIS system, which maintains a library of programs in which some parts remain abstract and undefined [38]. PARIS provides an interactive mechanism to search through the library for a schema that can be reused and supports the refinement and conversion of non-program abstract entities in the retrieved schema to concrete source programs.

SUMMARY OF SOFTWARE REUSE TECHNOLOGIES

A large number of approaches for identifying, classifying, retrieving, understanding, and integrating reusable resources are being actively researched. This research seems to be at a formative stage. There is a need to determine the most effective techniques in each of these phases of the reuse process and to develop an integrated and standardized approach that can be readily understood and adopted by a large community of software developers. Since reuse requires understanding and matching of requirements with stored resources, there is also a need to match the language of classification and retrieval to the language of the requirements specification.

ADVANTAGES OF SOFTWARE REUSE

Software reuse can save time, save money, and increase the reliability of resulting products. There often is a large body of software available for use on a new application, but the difficulty in locating the software or even being aware that it exists results in the same or similar components being re-invented over and over again. In order to overcome this impediment, a necessary first step is the ability to organize and catalog collections software components and provide the means for developers to quickly search a collection to identify candidates for potential reuse [39]. Software reuse is the use of existing software or software knowledge to construct new software [40]. Effective software reuse requires that the users of the system have access to appropriate components. Component is a well-defined unit of software that has a published interface and can be used in conjunction with components to form larger units [41].

Classifying software allows reusers to organize collections of components into structures that they can search easily. Most retrieval methods require some kind of classification of the components.

Less development time, and therefore cost, is necessary because there is a repository of software assets with which to start. Although time is required to assess the applicability of a given reusable asset to a new software system or product, that time is minimal in comparison to development time for a new module in the "one-time only" style [1].

CONCLUSION AND FUTURE WORK_

In this paper, the building of a code level reusable component has been discussed. The biggest advantage of the software reuse is that it reduces the time and energy in developing any software. Building code level reusable components will increase the quality and reduces time to design.

One of the best methods is to develop code level reusable components. These will give the best code reuse and improves the quality of the product. At its best code reuse is accomplished through the sharing of common classes and/or collections of functions, frameworks and procedures. This paper gives the concept of Reuse Code Levels and explores their applicability to reuse the software components. The code level reuse can save the time and money and increase the productivity and quality in the product.

In future, there is a plan to add more modules to the component, that provide with new rules and guidelines. Software reusable component as a function module is to check if optimized code is being used in building programs and applications by providing coding guidelines and standards, and also using best practices used for coding.

REFERENCES_

[1] B.Jalender, Dr A.Govardhan, Dr P.Premchand "A Pragmatic Approach To Software Reuse", 3 vol 14 No 2 Journal of Theoretical and Applied Information Technology (JATIT) JUNE 2010 pp 87-96.

[2] B.Jalender, Dr A.Govardhan and Dr P.Premchand. Article: Breaking the Boundaries for Software Component Reuse Technology. International Journal of Computer Applications 13(6):37–41, January 2011. Published by Foundation of Computer Science.

[3] Article "Considerations to Take When Writing Reusable Software Components"

[4]. R.G. Lanergan and C.A. Grasso, "Software Engineering with Reusable Designs and Code," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984, pp. 498-501

[5] J.M. Boyle and M.N. Muralidharan, "Program Reusability through Program Transformation," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984, pp. 574-588.

[6] T.J. Biggerstaff and A.J. Perlis, eds.," Software Reusability: Concepts and Models" ACM Press, New York, vol. 1, 1989.

[7] B.Jalender, Dr A.Govardhan, Dr P.Premchand, Dr C.Kiranmai, G.Suresh Reddy" Drag and Drop: Influences on the Design of Reusable Software Components" International Journal on Computer Science and Engineering Vol. 02, No. 07, pp. 2386-2393 July 2010.

[8] B. Jalender, N. Gowtham, K. Praveenkumar, K. Murahari, K. sampath"Technical Impediments to Software Reuse" International Journal of Engineering Science and Technology (IJEST), Vol. 2(11),p. 61366139.Nov 2010.

[9] W.A. Hegazy, The Requirements of Testing a Class of Reusable Software Modules, Ph.D. dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, June 1989.

[10]. Bollinger, T.B., and Pfleeger, S.L. The economics of reuse: issues and alternatives. Proceedings of the Eighth Annual National Conference on Ada Technology, 1990, pp. 436-447.

[11]. Isoda, S. Experience report on software reuse project: its structure, activities, and statistical results. Proceedings of the 14th Annual International Conference on Software Engineering, 1992, pp. 32~326.

[12]. Rubin, K. Reuse in software engineering: an object-oriented perspective. Proceedings of IEEE COMPCON, 1990, pp. 340-346.

[13] J. Favaro, K. Favaro, and P. Favaro, "Value Based Software Reuse Investment," Annals of Software Eng., vol. 5, pp. 5-52, 1998.

[14] Douglas Eugene Harms "The Influence of Software Reuse on Programming Language Design" The Ohio State University 1990.

[15]http://www.esdswg.com/softwarereuse/Resources/rrls/RRLs_v1.0.pdf.

[16] Department of the Navy. DON "Software Reuse Guide, NAVSO P-5234-2, 1995.

[17] "Breaking Down the Barriers to Software Component Technology" by Chris Lamela IntellectMarket, Inc

[18] D'Alessandro, M. Iachini, P.L. Martelli, "A The generic reusable component: an approach to reuse hierarchical OO designs" appears in: software reusability,1993

[19] Charles W. Krueger Software Reuse "ACM Computing Surveys (CSUR) Volume 24, Issue 2 (June 1992).

[20] Article "assess reuse risks and costs "www.goldpractice.thedacs.com/practices/arrc/". [21] M. Pat Schuler, "Increasing productivity through Total Reuse Management (TRM)," Proceedings of Technology2001: The Second National Technology Transfer Conference and Exposition, Volume 2, Washington DC, December 1991, pp. 294-300.

[21] M. Pat Schuler, "Increasing productivity through Total Reuse Management (TRM)," Proceedings of Technology2001: The Second National Technology Transfer Conference and Exposition, Volume 2, Washington DC, December 1991, pp. 294-300.

[22] Constance Palmer, "A CAMP update," AIAA-89-3144, Proceedings of Computers in Aerospace 7, Monterey CA, Oct. 3-5, 1989

[23]. Gaffney, J.E., Jr., and Durek, T.A. Software reuse---key to enhanced productivity: some quantitative models. Information Software Technology, 31,5 (June 1989),25&--267.

[24]. Poulin, J.S.; Caruso, J.M.; and Hancock, D.R. The business case for software reuse. IBM Systems Journal, 32,4 (1993), 567-594.

[25]. Boehm, B.W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[26] Constance Palmer, "A CAMP update," AIAA-89-3144, Proceedings of Computers in Aerospace 7, Monterey CA, Oct. 3-5, 1989

[27] Michael L. Nelson, Gretchen L. Gottlich, David J. Bianco, Sharon S. Paulson, Robert L. Binkley, Yvonne D. Kellogg, Chris J. Beaumont, Robert B. Schmunk, Michael J. Kurtz, Alberto Accomazzi, and Omar Syed, "The NASA Technical Report Server", Internet Research: Electronic Network Applications and Policy, vol. 5, no. 2, September 1995, pp. 25-36.

[28] Pamela Samuelson, "Is copyright law steering the right course?" IEEE Software, September 1988, pp. 78-86.

[29] Cai, M.R. Lyu, K. Wong, "Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes," in Proceedings of the 7th APSEC, 2000

[30] Jihyun Lee, Jinsam Kim, and Gyu-Sang Shin "Facilitating Reuse of Software Components using Repository Technology" Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03).

[31] Ralph E. Johnson & Brian Foote "Designing Reusable Classes" (IEEE Computer Society Press Tutorial) IEEE Computer Society Press, Los Alamitos, CA May 1991, 299 p.

[32] http://diwt.wordpress.com/tag/struts2/

[33] P. Shireesha, S.S.V.N.Sharma,"Building Reusable Software Component For Optimization Check in ABAP Coding" International Journal of Software Engineering & Applications (IJSEA) Vol.1, No.3, July 2010

[34]. Paulk, M.C.; Curtis, B.; and Chrissis, M.B. Capability Maturity Model for Software. Pittsburgh, PA: Software Engineering Institute, CMUlSEI-91-TR-24, 1991.

[35]. Karlsson, E., ed. Software Reuse: A Holistic Approach. New York: John Wiley, 1995.

[36]. Timothy G. Olson; Linda Parker Gates; Julie L Mullaney; James W. Over; Neal R. Reizer; Marc 1. Kellner; Richard W. Phillips; Salvatore J. DiGennaro A Software Process Framework for the SEI Capability Maturity Model: Repeatable Level, June 1993.

[37]. Kernighan, B.W. The Unix system and software reusability IEEE Transaction on S~ftware Engineering, 10,5 (September 1984), 513-518.

[38]. Katz, S.; Richter, C.H.; and The, K. PARIS: a system for reusing partially interpreted schemas. Proceedings ~f IEEE 9th International Conference on S~ftware Engineering, 1987, pp.377-385.

[39] Douglas Eugene Harms "The Influence of Software Reuse on Programming Language Design" The Ohio State University 1990.

[40] B.H. Liskov and S.N. Zilles, "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, vol. SE-1, no. 1, March 1975, pp. 7-19.

[41] J.M. Boyle and M.N. Muralidharan, "Program Reusability through Program Transformation," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984, pp. 574-588.

AUTHORS' PROFILE LINK_

https://github.com/Zulkarnine43