TECHNICAL IMPEDIMENTS TO SOFTWARE REUSE

B.JALENDER [1], N.GOWTHAM [2], K.PRAVEEN KUMAR [3], K.MURAHARI [4], K.SAMPATH [5]

[1] Asst Professor, Department of IT, VNRVJIET, Hyderabad, India-500090

[2] Asst Professor, Department of CSE, KITS HUZURABAD, KARIMNAGAR,AP-505468.

[3] Associate Professor, Department of CSE, KITS HUZURABAD, KARIMNAGAR,AP-505468.

[4] Software Engineer, Symantec Software and Services India Pvt. Ltd, Chennai, India-600035.

[5] Research Scholar, Department of CSE, JNTU kakinada, India-533004.

Abstract:
A good software reuse process facilitates the increase of productivity, quality, reliability, and the decrease of costs and implementation time. One of major impediments to realizing software reusability in many organizations is the inability to locate and retrieve existing software components. 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. This paper addresses the technical impediments to software component reuse technology.

Keywords: Reuse; component; impediments; software.

1. Introduction

1.1 What Is A “Reusable Software Component?”

Reusable software refers to software components that can be incorporated into a variety of programs without modification. Reusable software components are designed to apply the power and benefit of reusable, interchangeable parts from other industries to the field of software construction. Other industries have long profited from reusable components [1]. Reusable electronic components are found on circuit boards. A typical part in your car can be replaced by a component made from one of many different competing manufacturers. Lucrative industries are built around parts construction and supply in most competitive fields. The idea is that standard interfaces allow for interchangeable, reusable components [2]. This definition of reuse does not meet our definition because it is not concerned with reusable software components incorporated into client programs.

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

1.2 Advantages of Software Reuse

One of major impediments to realizing software reusability in many organizations is the inability to locate and retrieve existing software components. 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 [14].

Software reuse is an important area of software engineering research that promises significant improvements in software productivity and quality [4]. Software reuse is the use of existing software or software knowledge to construct new software [11]. Effective software reuse requires that the users of the system have access to appropriate components. The user must access these components accurately and quickly, and be able to modify them if necessary. 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 [3]. Reuse deals with the ability to combine separate independent software components to form a larger unit of software. To incorporate reusable components into systems, programmers must be able to find and understand them. 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.

There are several reasons why designing and building reusable software components potentially improves both software quality and programmer productivity. First, because the cost of designing and building a component can be amortized over many uses, it is economically feasible to commit the time, intellectual energy, and money to do things right the first time. Committing necessary resources during the appropriate stages of the component's lifecycle improves quality in an obvious way [6].

Second, the designer of a reusable part knows that it will be used in applications unimaginable at the time of design, and will likely take the job seriously and design a quality part. He or she will probably take the time to look at the larger picture, imagine and anticipate uses and variations of the part, and make the design general by factoring out idiosyncrasies of specific applications. In addition, the psychological effect of knowing one's design will be scrutinized by future programmers can have a positive impact on the quality of the part's design.

Third and perhaps most important, programmer productivity will increase because it is usually easier to reuse a well-designed software component than to design and implement one on the fly. This proposition is considered dubious by some who envision only very simple components as reusable, but for components with complex behavior it is quite obvious. However, this claim does not seem to have been established by experimental evidence yet, due partly to the near-absence of truly reusable software components [8].

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].

2. Non-Technical Impediments to Software Reuse

First non-technical impediment is an economic. Producing and selling software is the business of most of the IT and Software companies. If a Software company sells a truly bugs free and reusable component to a customer, that customer may no longer need the services of the software company. In order to making reusable software economically profitable the price of the reusable components should decrease and giving rewards to the manufacturers of reusable components with out eliminating the economic market for the components[14].

The second non technical impediment is an organizational impediment. The detailed catalogs describing the available components must be provide to the potential customers by the manufacturers. Customer must be able to efficiently search these catalogs and easily determine whether a particular reusable component is appropriate for a particular application.Negative psychological effects are another non-technical impediment to software reuse [5].

Trying to apply one-dimensional technical solutions to complex software development problems is an exercise in frustration and a recipe for costly project failures. For instance, attempting to translate software implementations entirely from high-level SDL specifications or from abstract ``analysis rules'' rarely succeeds for complex networked applications. Likewise, using the latest design methodology, modeling notation, programming language, or middleware technology fads can't guarantee success [5].

The urge to apply one-dimensional solutions to complex problems isn't limited to technologists, however. For instance, there is a school of thought that claims only the non-technical impediments to reuse are worth addressing since systematic reuse fails solely for economic and organizational reasons, not technological ones. According to this perspective, investing in education or training to improve the technical skills of developers is pointless because it has no impact on success [14].

One-dimensional non-technical solutions are no better than one-dimensional technological solutions. Managerial and organizational support is certainly desirable and compulsory for large-scale adoption of systematic reuse across an enterprise. Moreover, focusing solely on organizational and economic impediments at the expense of technology and skills-building, can yield a corporate culture of ``learned helplessness." Developers suffering from this malady often postpone improving their design and reuse skills until the entire organization is cured. This approach is as fulfill all the customer requirements to solidify before engaging in architecture and design phases. Failing to invest in technology and education can greatly hamper a company's ability to compete effectively, particularly when time-to-market is crucial to success [14].

So believe that we must not wait passively for organizational and economic impediment problems to be resolved completely before building the technical skills and experience level of developers. Instead, we must initiate and support skills-building education now and sustain them over time. These skills are ultimately required to succeed with systematic reuse, in particular and high-quality software development, in general [13].

3. Technical Impediments to Software Reuse

There are several technical issues that currently keep reusable software from becoming a reality. In a very real sense solutions to these technical impediments are more important than solutions to the non technical ones, for until it is technically possible to design and build truly reusable components, management and organization cannot achieve widespread reuse of software. The first technical impediment is the lack of formal specifications for components. A programmer cannot be expected to reuse an existing part unless its functionality is crystal-clear. All too often programmers “reinvent the software wheel” because the functionality of existing parts is unclear or vague, and the alternatives deciphering source code and trial-and-error testing are often more painful than simply starting from scratch[14].

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. Also, the principles of information hiding and abstraction should be followed, so providing the client with source code of the component [7] is not acceptable.

A second technical impediment is the inability to certify the correctness of a component. Of course, attempting to certify the correctness of a component whose specification is incomplete or ambiguous is an exercise in futility. Obviously the problem of formal specification must be solved before this issue can be meaningfully addressed. However, even with formal specification the problem of certification (i.e., formal verification) is difficult, in part because the techniques are still being developed and have generally not been applied to large programs with complex data structures [14]. Also, many programming languages have constructs (such as aliasing) that significantly complicate program verification [8] [9]. Testing, a weaker certification method, has received much attention recently [10].

Organizational impediments - e.g., developing, deploying, and supporting systematically reusable software assets requires a deep understanding of application developer needs and business requirements. As the number of developers and projects employing reusable assets increases, it becomes hard to structure an organization to provide effective feedback loops between these constituencies [13].

Economic impediments -- e.g., supporting corporate-wide reusable assets requires an economic investment, particularly if reuse groups operate as costcenters. Many organizations find it hard to institute appropriate taxation or chargeback schemes to fund their reuse groups [13].

Administrative impediments -- e.g., it's hard to catalog, archive, and retrieve reusable assets across multiple business units within large organizations. Although it's common to scavenge small classes or functions opportunistically from existing programs, developers often find it hard to locate suitable reusable assets outside of their immediate workgroups [13].

As if these non-technical impediments aren't daunting enough, reuse efforts also frequently fail because developers lack technical skills and organizations lack core competencies necessary to create and/or integrate reusable components systematically [11]. For instance, developers often lack knowledge of, and experience with, fundamental design patterns in their domain, which makes it hard for them to understand how to create and/or reuse frameworks and components effectively [11].

Another technical impediment is the relatively poor performance of reusable parts. Part of the problem here is the assumed trade-off between generality and performance that most programmers believe exists. In fact, there is no theoretical basis for this belief, although empirical evidence seems to support it. The problem is that most parts classified as reusable were designed and implemented using classical data structures and algorithms as taught in introductory computer science classes. These components, however, were not designed to be reusable, and performance suffers as a result. New evidence suggests that reusable parts can be designed that exhibit no significant performance degradation relative to non-reusable custom parts [12].

Category	**Research Issues**	**Impediments to software reuse**
General Issues	Definition and Scope	The lack of well understood and accepted terminology to Describe concepts
	Economic Issues	The investment needed to promote software reuse. The lack of an economic model to explain the benefits and Costs of software reuse.
Technical Issues	Software Reuse Process	The lack of a methodology for creating and implementing Software reuse.
	Software Reuse Technologies	The lack of reusable and reliable software resources ,The lack of tools and techniques for supporting software reuse
Non-technical Issues	Behavioral Issues	The lack of commitment, encouragement, training and rewards for software reuse
	Organizational Issues	The lack of organizational support to institutionalize software reuse .The difficulty in measuring the gains from reuse.
	Legal and Contractual Issues	Intellectual property rights and contractual problems of Software reuse

Table [1]. Impediments to software reuse

4. Conclusion and Future Work

Software quality and programmer productivity are two of the biggest challenges facing the software engineering community. Reusability, a mainstay of other engineering disciplines, is an approach to software development that addresses both of these issues [13]. A designed-for-reuse software component is economically efficient to design and build, it most likely is of a higher quality than a “scavenged” part, and reusing it increases the productivity of client programmers. Despite these advantages, there are both technical and non-technical impediments to widespread software reuse. It substantially overcomes the architectural impediments that have hindered some previous large-scale reuse attempts. It appears to represent significant progress towards realizing the promise of rapid software development through integration of large-scale, reusable application components [14].

5. 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]. 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
[3] 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.
[4] T.C. Jones, “Reusability in Programming: A Survey of the State of the Art,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 488-494.
[5] *Software Reusability: Concepts and Models*, T.J. Biggerstaff and A.J. Perlis, eds., ACM Press, New York, vol. 1, 1989.
[6] C.A.R. Hoare, “Hints on Programming Language Design,” In *Programming Languages: A Grand Tour*, E. Horowitz, ed., Computer Science Press, Rockville, MD, pp. 31-40, 1983,
[7] J. Krone, *The Role of Verification in Software Reusability*, Ph.D. dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, August 1988.
[8] H.D. Mills, M. Dyer, and R.C. Linger, “Cleanroom Software Engineering,” *IEEE Software*, vol. 4, no. 5, September 1987, pp. 19-25.
[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] D.L. Parnas, “A Technique for Software Module Specification with Examples,” *Communications of the ACM*, vol. 15, no. 5, May 1972, pp. 330-336
[11] 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.
[12]Sullivan,K.J.;Knight,J.C.;“Experience assessing an architectural approach to large-scale, systematic reuse,” in *Proc. 18th Int’l Conf. Software Engineering*, Berlin, Mar. 1996, pp. 220–229
[13] Schmidt, D. C., *Why Software Reuse has Failed and How to Make it Work for You* [Online], Available: http://www.flashline.com/content/ DCSchmidt/lesson_1.jsp, [Accessed: 18 August 2002].
[14] Douglas Eugene Harms “The Influence of Software Reuse on Programming Language Design” The Ohio State University 1990.