Humanizing Efficiency and Software Superiority through Concept of Reusability

Dr. Shailendra Narayan Singh

Department of Computer Science and Engg.
Amity School of Engineering and Technology
Amity University Uttar Pradesh

ABSTRACT: Work in software reuse focuses on reusing Objects. In this context, finding a reusable objects are driven by a desired functionality. This paper proposes a change to this common view. Software reuse has become a topic of much interest in the software community due to its potential benefits, which include increased product quality and decreased product cost and schedule. The most substantial benefits derive from a product line approach, where a common set of reusable software assets act as a base for subsequent similar products in a given functional domain. The upfront investments required for software reuse are considerable, and need to be duly considered prior to attempting a software reuse initiative.

Keywords: Software reuses, Domain Engineering, Frame working, Assets, Repository knowledge.

1. Introduction

Software reuse has been a lofty goal for Software Engineering (SE) research and practice, as a means to reduced development costs and improved quality. The past decade has seen considerable progress in fulfilling this goal, both with respect to research ideas and industrial practices. Systems fail because of inadequate performance, security, reliability, usability, or precision, to name a few. Quality concerns, therefore, should also be front and centre in methods for software reuse. Software reuse is the process of implementing or updating software systems using existing software assets. [1] Although first reaction may lead to believe that a "software asset" is simply another term for source code, this is not the case. Software assets, or components, include all software products, from requirements and proposals, to specifications and designs, to user manuals and test suites. Anything that is produced from a software development effort can potentially be reused. 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.

2. Obstacles in Reusing Software Historically

Many developers have successfully applied reuse opportunistically, e.g., by cutting and pasting code snippets from existing programs into new programs. Opportunistic reuse works fine in a limited way for individual programmers or small groups. Like many other promising techniques in the history of software. There have certainly been successes, e.g., sophisticated frameworks of reusable components are now available in OO languages running on many OS platforms. In general, however, these frameworks have focused on a relatively small number of domains, such as graphical user-interfaces or C++ container libraries. Moreover, component reuse is often limited in practice to third-party libraries and tools, rather than being an integral part of an organization's software development processes. In experience, non-technical impediments to successful reuse commonly include the following:

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.

Economic impediments -- e.g., supporting corporate-wide reusable assets requires an economic investment, particularly if reuse groups operate as cost-center. Many organizations find it hard to institute appropriate taxation or charge-back schemes to fund their reuse groups.

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.

Political impediments -- e.g., groups that develop reusable middleware platforms are often viewed with suspicion by application developers, who resent the fact that they may no longer be empowered to make key architectural decisions. Likewise, internecine rivalries among business units may stifle reuse of assets developed by other internal product groups, which are perceived as a threat to job security or corporate influence.

Psychological impediments -- e.g., application developers may also perceive "top down" reuse efforts as an indication that management lacks confidence in their technical abilities. In addition, the "not invented here" syndrome is ubiquitous in many organizations, particularly among highly talented programmers.[2]

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. 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.

3. Systematic Reuse of Software’s

Domain engineering is the key concept and focus of current reuse efforts. The prospect of being able to reuse entire quality subsystems without change, especially at today’s business speed of “we needed it yesterday”, is a significant gain to both customers and software organizations.

Systematic software reuse is a promising means to reduce development cycle time and cost, improve software quality, and leverage existing effort by constructing and applying multi-use assets like architectures, patterns, components, and frameworks.

Horizontal reuse refers to software components used across a wide variety of applications. In terms of code assets, this includes the typically envisioned library of components, such as a linked list class, string manipulation routines, or graphical user interface (GUI) functions. Horizontal reuse can also refer to the use of a commercial off-the-shelf (COTS) or third-party application within a larger system, such as an e-mail package or a word processing program. A variety of software libraries and repositories containing this type of code and documentation exist today at various locations on the Internet.

Vertical reuse, significantly untapped by the software community at large, but potentially very useful, has far reaching implications for current and future software development efforts. The basic idea is the reuse of system functional areas, or domains that can be used by a family of systems with similar functionality. Domain engineering is “a comprehensive, iterative, life-cycle process that an organization uses to pursue strategic business objectives. It increases the productivity of application engineering projects through the standardization of a product family and an associated production process.”[3] Which brings us to application engineering, the domain engineering counterpart: “Application engineering is the means by which a project creates a product to meet a customer’s requirements. Unfortunately, software reuse doesn’t just happen. Ad hoc reuse, (i.e., reusing a function here, a function there, often times with modifications), also known as opportunistic reuse, doesn’t reap the same large-scale benefits as a domain engineering approach. And it’s not just a technical issue; it is highly managerial in nature. As much as libraries of reusable code and other assets are important, they will not be fully utilized without management and process support of reuse.

4. Fundamentals for Creating Reusable Software

Software reuse is most effective when the following prerequisites are met:

The market is competitive -- In a competitive business environment, such as financial services or wireless networking, time-to-market is crucial. It’s therefore essential to leverage existing software to reduce development effort and cycle time. When a market is not competitive, however, organizations tend to reinvent, rather than reuse, software.

The defense industry in the 1980’s is a good example of the “reinvent rather than reuse” phenomenon. During the defense buildup in the Reagan administration, defense-related R&D funding was abundant. Not surprisingly, little systematic reuse was achieved, even though the Ada programming language was designed explicitly to support software reuse. Starting in the early 1990’s, however, the substantial decrease in DoD funding had a powerful impact on the defense industry. Contemporary aerospace and defense contractors now aggressively seek to integrate “commercial-off-the-shelf” (COTS) software and hardware in order to remain competitive in their changing marketplace.

The application domain is complex -- Components that are relatively easy to develop, such as generic linked lists, stacks, or queues, are often rewritten from scratch rather than reused. In contrast, developers working in highly complex domains, such as distributed, real-time systems are often willing to reuse components, such as dynamic scheduling frameworks when building equivalent solutions from scratch proves too error-prone, costly, or time-consuming.

The corporate culture and development process are supportive -- Not only is it hard to develop high-quality reusable components and frameworks, it’s even harder to reap the benefits of reuse immediately. Significant investment must be expended up-front to produce efficient, flexible, and well-documented reusable software assets before they can be leveraged in subsequent generations of a product line. Therefore, organizations must support an appropriate software development process that allows systematic reuse to flourish. Ideally, an organization’s software process should reward developers who invest the time and effort to build, document, and reuse robust and efficient components. For instance, a reward system could be built into project budgets, with incentives based on the number of software components reused by individuals or groups. I still find companies, however, whose processes measure programmer productivity solely in terms of the number of lines of source code written from scratch, which penalizes developers who attempt to reuse existing software.

Organization and Process- A strong organizational foundation must exist for reuse to succeed, since domain engineering involves a different way of looking at software products, called a product line approach. A
group that is responsible for the maintenance of the reuse infrastructure must exist. On each project, responsibility must be assigned for the acquisition and maintenance of reusable components for the project.

**Technical Expertise:** Transferring to a product line approach requires some different technical skills than traditional software development processes, along with many of the current familiar techniques, such as layered architectures, object-oriented programming, information hiding, and abstract interfaces, to name a few. One "new" addition, an aspect of domain engineering, is domain analysis, which involves producing a domain model of the product line that identifies common members and allowable variations for each.

**Attractive "reuse magnets" exist** -- To attract systematic reuse, it crucial to develop and support "reuse magnets," [4] i.e., well-documented framework and component repositories. These repositories must be well-maintained so that application developers will have confidence in their quality and assurance that any defects they encounter will be fixed promptly. Likewise, framework and component repositories must be well-supported so that developers can gain experience through hands-on training and mentoring programs.

In the experience, "open-source" development processes are an effective process for creating attractive reuse magnets. Open-source processes have yielded many widely used software tools and frameworks, such as Linux, Apache, GNU, ACE, and TAO. The open-source model allows users and developers to participate together in evolving software assets. One of the key strengths of this model is that it scales well to large user communities, where application developers and end-users can assist with much of the quality assurance, documentation, and support.

### 5. Existing tools for Reusability

Key for successful reuse is the organization and accessibility of the common reusable assets. Asset management tools, such as repositories, for architectures, designs, documentation, and code must be developed and maintained. Also needed are tools to aid in the integration of architecture, design, and software products, in order to speed prototyping, full-scale development, modifications, and maintenance. Along with these tools, a strong configuration management process must be in place to work with the architecture team and track the evolution of the product line. "Automated browsing tools with sufficient sophistication must be acquired or developed to facilitate search and retrieval. After all, if the users cannot find the asset, they won't use it, and the investment in the repository has been wasted. Configuration management tools must be incorporated into asset repositories in order to trace an asset to the systems in which it was used. This type of information assists future users of an asset in deciding its appropriateness to their situation." The tight integration of configuration management activities with the reusable assets assures the validity of the common core, another definite must while developing with reusable assets.

By far the most important part of the reuse process is the people. If the people in the organization do not understand the concepts behind reuse, and do not see the benefits, reuse won't happen. Since software reuse is not a common standard, staff training and subsequent buy-in must be accomplished for a reuse effort to succeed.

In the first place, staff must be presented with the principles of reuse, and the long-term benefits. Awareness is the first step. Once the basic concepts are understood, guidelines and procedures for creating and retrieving reusable assets must be presented. But people must have incentive. In this day and age, when software professionals change companies every two or three years, long term benefits for a company are not enticing reasons to spend more time and care on modules to make them reusable. So short term benefits and rewards must be available to individuals who contribute to the reuse initiative by creating reusable assets or reusing assets from the library. Developing modules with the rigorous standards required for reuse takes about twice as long as one-time system modules, and that extra effort should be supported by management through rewards and recognition. Staff members will quickly see benefits to software reuse, and reuse will become more popular throughout the organization.

### 6. Conclusion

As the saying goes, "no pain, no gain," and the reuse of software is no exception. The product line approach to software reuse requires substantial upfront investment with substantial, but not immediate, benefits. Much commitment, planning, and effort are required to begin a reuse program. Reuse processes and procedures must be incorporated into the existing software development process. Repositories of software assets must be created and maintained. Reusable assets must be designed for reusability. People must be trained in the skills of software reuse. Despite the initial overhead, there are high benefits to software reuse, if appropriate processes are invoked and the requisite planning takes place. Product quality and reliability can increase. Project development time can decrease, along with associated project costs. Project scheduling can become another standard calculation instead of a guesstimate. All these benefits, in the long term, can dramatically increase productivity in an organization, and decrease the overall risk of project development by supplying a solid foundation from which all subsequent product family members are derived.

### REFERENCES

AUTHOR

Dr. Shailendra Narayan Singh working in Amity University Uttar Pradesh and having experience in teaching and administration. He did his B. Tech, M. Tech, Ph.D. in the field of Computer Science and Engineering. His research area is software reusability for aspect oriented and object oriented software. Dr. Shailendra Narayan Singh is having more than fifteen years of teaching experience in the academic field of computer science and engineering in Indian and Abroad Universities. Dr. S N. Singh is author of four book titles Operating System, Fundamentals of Internet, Software Project Management Part-I and Part-II. Teaching is his first love and participates in various conferences, seminars in National and International.