www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 11 November, 2013 Page No. 3217-3222

A Realistic Approach to Systematic Reuse

Sampath Korra[1], Dr S.Viswanadha Raju[2], Dr A.Vinaya Babu[3]

[1] Research Scholar, Department of CSE, JNTUK, Kakinada.AP, INDIA.

[2] Professor, JNTUH College of Engineering, Jagtial, Karimnagar, AP, INDIA.

[3] Principal, Department of CSE, JNTUH, Hyderabad,A.P, INDIA.

ABSTACT:

Every software project practices some kind of reuse as a common sense practice. Very often, practitioners use parts of code, documents and experiences from previous projects as a personal initiative. Of course, this kind of reuse brings some benefits to the company. However, it is generally performed in isolation from other projects, depends on the individual's initiative and has very limited impact.

Systematic Software Reuse is the capability of an organization to obtain maximum profit from the experiences acquired in former projects by identifying the reuse opportunities a priori and establishing the appropriate organizational, managerial and budgetary support. Software Reuse reduces development costs and time by avoiding the duplication of work.

Keywords: reuse, objects, development, application.

1. INTRODUCTION

Although computing power and network bandwidth have increased dramatically in recent years, the design and implementation of networked applications remains expensive and error-prone. Developing software that achieves these qualities is hard; systematically developing high quality *reusable* software components and frameworks is even harder **[5]**. Reusable components and frameworks are inherently abstract, which makes it hard to engineer their quality and to manage their production.

During the past decade, worked with hundreds of telecommunication, aerospace, and medical companies and written millions of lines of code while developing widely reusable middleware components and frameworks **[10,7]** for networked applications. It also had the opportunity to document several dozen patterns **[9]** and architectures **[6]** that guide the design of these components and frameworks. In addition, It have taught hundreds of tutorials and courses on these topics for thousands of developers and students. In spite of formidable non-technical and technical challenges, It is identified a solid body of knowledge, experience, and software artifacts that can significantly enhance the systematic reuse of networked application software.

In this paper, outline of the paper is common reasons why the systematic reuse has failed in the past. Then discuss proven steps that organizations, projects, and individuals can take to avoid these traps and pitfalls.

1.1 Systematic reuse needs a systematic approach

By systematic reuse, we mean an institutionalized organizational approach to product development in which reusable assets are purposely created or acquired, and then consistently used and maintained to obtain high levels of reuse, thereby optimizing the organization's ability to produce quality software products rapidly and effectively.

This requires a significant effort to change culture, organization, and a multitude of other factors. These changes are quite radical, and more widespread than those incremental and ongoing changes associated with CPI (continuous process improvement). Reuse is a business issue. It must be viewed as an organizational asset, to be invested in, improved, and leveraged effectively and consistently[14]

Often, sweeping changes in the software development organization will be needed to institute large-scale, systematic reuse. Business changes fund product family design and construction with the goals of improved time to market and cost for several related products, investing in reusable assets. Software process changes manage and use a reusable asset repository. Many cultural and management changes create new, unfamiliar roles. The magnitude of the changes, and the issues encountered, are similar to those encountered when doing Business Process Reengineering. We find that the systematic methods and skills used to design and implement such changes can also be applied to software organizations.[17]

2. WHY SYSTEMATIC REUSE HAS FAILED

In theory, organizations recognize the value of systematic reuse and reward internal reuse efforts. 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.

It is observed that developers often put too much faith in language features, such as inheritance, polymorphism, templates, and exception handling, as the primary means to foster reuse. Unfortunately, languages alone don't adequately capture the commonality and variability of abstractions and components required to build and systematically apply reusable software in complex application domains.

3. HOW TO MAKE SYSTEMATIC REUSE SUCEESS

Although the track record for systematic software reuse has been rather spotty historically, several key trends bode well for software reuse in the future:

- Component- and framework-based middleware technologies, such as CORBA, J2EE, and .NET, have become mainstream.
- An increasing number of developers of projects over the past decade have successfully adopted OO design techniques, such as UML and patterns, and OO programming languages, such as C++, Java, and C#.

These trends are particularly evident in markets, such as electronic commerce and data networking, where reducing development cycle time is crucial to business success.

Over the past decade, It has worked with many companies, including Boeing, Cisco, Ericsson, Iridium, Kodak, Lucent, Motorola, SAIC, Siemens, and Sprint, building reusable networked applications using OO design techniques and programming languages **[9,10]**. These projects have applied a range of reusable middleware tools including CORBA, the ACE framework **[10]**, which is a C++ framework that implements many patterns for concurrent networked applications, and TAO **[7]**, which is a high-performance, real-time implementation of CORBA that leverages the framework components in ACE.

Prerequisites for Successful Systematic Reuse

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. Still it is 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.

1. *Attractive ``reuse magnets'' exist* -- To attract systematic reuse, it crucial to develop and support ``reuse magnets," **[3]** *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 addition, ``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 **[11]**.

   Moreover, open-source development efforts tend to have short feedback loops between the point when a bug is discovered and the bug is fixed. This increases the incentive for the user community to help with the debugging process since they are ``rewarded" by rapid feedback and fixes once bugs are identified. In addition, because the source code is available for inspection, developers in the user community can often help fix any bugs they find which further amortize the overall debugging effort and improve software quality rapidly.

2. *Strong leadership and empowerment of skilled architects and developers* – It has been observed that the ability of companies and projects to succeed with reuse is highly correlated with the quality and quantity of experienced developers and effective leaders. Conversely, reuse projects that lack a critical mass of developers with the necessarily technical and leadership skills rarely succeed, regardless of the level of managerial and organizational support.

   In general, the level of experience required to succeed with systematic reuse depends largely on whether programmers are trying to develop reusable components or to use them. It is found that *developing* reusable frameworks and components for complex domains, such as telecom or avionics, requires highly experienced and skilled architects and developers. These individuals must be trained and empowered to create, document, and support horizontal middleware platforms that reduce the effort required to develop vertical applications.

   In general, the more complex the domain, the greater the skills and leadership required to develop effective reusable middleware that can encapsulate complex communication protocols and mechanisms for concurrency, locking, persistence, fault tolerance, connection management, event demuxing, and service configuration. When middleware architects and developers are successful, they create component abstractions that hide these error-prone and tedious mechanisms and protocols. Application developers therefore needn't be as experienced with complex systems-level technologies since they can program to these higher-level component abstractions.

   It is observed, however, that horizontal middleware platform efforts generally fail when application developers are (1) too inexperienced, (2) the domain is sufficiently challenging, and (3) the middleware team lacks sufficient training, resources, time, or empowerment to create a stable platform. It's therefore important that developers at all levels improve their technical skills and learn how to apply good software principles, patterns, and practices.

Unfortunately, many organizations lack the mainly two prerequisites described above. As a result, these organizations often fall victim to the ``not-invented-here" syndrome and redevelop many software components from scratch. However, deregulation, global competition, and the general dearth of experienced application and middleware developers is making it increasingly hard to succeed by building complex networked applications from the ground up.

Maintain a Close Feedback Loop between Middleware Developers and Application Developers

Most useful middleware components and frameworks originate from solving real problems in particular application domains, such as telecommunications, medical imaging, avionics, or electronic commerce. A time-honored way of producing effective reusable middleware, therefore, is to *generalize* and *refactor* them from working systems and applications.

In contrast, reuse efforts that try to work top-down, *e.g.*, from high-level domain analysis, are *highly* likely to fail. The culprit is often the lack of close feedback loops between developers of reusable middleware and developers of applications. For instance, It has been observed that it's usually counter-productive to create reuse teams that build middleware frameworks and components in complete isolation from application teams.

Since reuse efforts rarely have sufficient resources to please all possible customers, it's important to be goal-directed, rather than exhaustive, in determining which assets to develop, enhance, and maintain. Without intimate feedback from application developers, therefore, the software artifacts produced by component teams rarely address core business problems and won't be reused effectively.

It's also important that the relationship between application developers and reuse groups be mutually synergistic. For instance, reuse teams should be responsive to fix problems that inevitably arise in their middleware. Likewise, application developers should actively help to improve reusable artifacts, rather than waiting passively for the reuse team to find and fix all the problems. While it's possible to depend upon developer altruism and good will to achieve these goals, it's usually more effective to institutionalize a reward system with incentives to encourage effective relationships between developers and users.

Buy, Rather than Build, System Infrastructure and Middleware Integration Frameworks

Frameworks can be classified according to their *scope*, as follows **[8]**:

- *Middleware integration frameworks* -- These frameworks are commonly used to integrate networked applications and components. Middleware integration frameworks are designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment. There is a thriving market for middleware integration frameworks, which are rapidly becoming commodities. Common examples include CORBA, J2EE, .NET, and transactional databases.

- *Enterprise application frameworks* -- These frameworks address vertical application domains, such as avionics mission computing, call processing, and manufacturing, and are the cornerstone of enterprise business activities. Relative to system infrastructure and middleware integration frameworks, enterprise frameworks are expensive to develop and/or purchase.

- In general, system infrastructure and middleware integration frameworks focus largely on internal software development concerns. Although these frameworks are essential to creating high quality software rapidly and cost-effectively, they typically don't generate substantial revenue. It's therefore often more cost effective to buy system infrastructure and middleware integration frameworks rather than build them in-house.

Develop Software Based on Architectures, Rather Than on Particular Middleware Technologies

It's very risky to expect that industry standards, such as CORBA, J2EE, or .NET middleware, will eliminate the complexity of developing networked applications. No single middleware technology is a panacea. Moreover, industry standards for middleware are not ubiquitous, nor are they implemented consistently.

For large-scale, long-lived networked applications it's essential to design and use *architectures* that transcend any specific technology or middleware standard. For instance, programming directly to a proprietary middleware API is risky since these APIs can rapidly become obsolete. Therefore found it's more fruitful to develop networked applications based on a common architecture that can be instantiated on a range of middleware and OS platforms.

An effective pattern seen applied repeatedly to organize a common software architecture is to use (1) *frameworks* in the horizontal middleware platform layers and (2) *components* in the vertical application layers. Components are self-contained instances of abstract data types (ADTs) that can be plugged together to form complete applications. Common examples of components include COM+ controls and CORBA Object Services.

The relationship between frameworks and components is highly synergistic, with neither subordinate to the other. Frameworks can be used to develop components, whereby component interfaces provide *Facades* for internal class structures inside a framework. Moreover, components can be used as ``pluggable strategies" within a framework.

Compared with frameworks, components are less tightly coupled and can support binary-level reuse more readily. For example, application developers can reuse components without having to subclass from existing base classes. In addition application developers are generally more comfortable and successful programming with components than they are customizing frameworks. Conversely, frameworks are useful for middleware teams because they help to simplify the development of horizontal platform software. Naturally, components can also be used to develop infrastructure and middleware.

Avoid One-dimensional ``Solutions'' Complex Software Development Problems

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.

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.

Unfortunately, one-dimensional non-technical solutions are no better than one-dimensional technological solutions. Managerial and organizational support is certainly desirable and essential for large-scale adoption of systematic reuse across an enterprise. In addition however, this support is not sufficient, nor even always necessary, to succeed with systematic reuse, particularly within smaller parts of organizations.

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 futile as waiting for *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. It can also cause companies to become dangerously out of touch with contemporary software practice, where an increasing number of companies *are* in fact succeeding with systematic reuse and COTS technology adoption. Not surprisingly, many of the large companies that suffered the most during the economic downturn in 2001 were also companies that most strongly resisted adopting systematic reuse and COTS.

Therefore it is believed that we must not wait passively for organizational and economic problems to be resolved completely before building the technical skills and experience level of developers. Instead, It 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.

Respect and Reward Top-Notch Developers and Architects
Developing robust, efficient, and reusable networked applications requires teams of people with a wide range of skills. It need experienced managers who know how to properly evaluate risks and opportunities in order to navigate their teams through the constantly changing landscape of technology and business drivers. Likewise, It need expert analysts and designers who have mastered design patterns, software architectures, and communication protocols in order to alleviate the inherent and accidental complexities of networked applications. Naturally, It also need seasoned programmers who can implement these patterns, architectures, and protocols to form reusable frameworks and components.

In practice, of course, it's hard to find top-notch software developers. Ironically, many companies---particularly large ones---still treat their developers as interchangeable, ``unskilled labor," who can be replaced easily. The increasingly noticing, however, that companies who respect and reward their top-notch software developers consistently outperform those who don't.

Systematic reuse is largely a by-product of good designs and experienced developers. Education is crucial to help improve developers' design skills. Fortunately, developing good reusable software requires many of the same set of skills, such as knowledge of architectures, patterns, frameworks, and components, necessary to develop good software in general. The time and effort spent on education will pay off therefore, whether or not developers actually write reusable software artifacts.

Keep the Faith
The repeatedly witnessed organizations that initiate systematic reuse efforts with the best of intentions, only to lose faith when various impediments arise or schedules slip. Inevitably, they then fall back onto familiar processes, *i.e.*, developing their software from scratch. It has been observed that reuse-in-the-large is best achieved when development and management leaders are unwavering and evangelistic.

Ultimately, organizations that attempt systematic reuse without providing an incubation environment will lose their faithful. Many of these faithful will be the most experienced developers or those most capable of coming up to speed quickly. In markets driven by ``Internet cycle times," the loss of valuable developers can devastate an organization's long-term competitive viability.

Keeping the faith requires keeping abreast of external R&D developments and global technology trends. In my travels throughout the software industry,

4. CONCLUSION AND FUTURE WORK

Over a decade's worth of experience developing and deploying reusable networked application artifacts has taught the importance of increasing developers' knowledge of patterns, as well as improving their skills at creating and supporting reusable components and frameworks. For more information on using patterns to build reusable networked application frameworks and components like CORBA, ACE, and TAO.

It has to stress, however, that these technological solutions alone are not silver bullets **[2]**. Firmly believe the promise of systematic reuse for networked applications will not be fully realized until it address both technical and non-technical impediments effectively. However, that there's no virtue in waiting for organizational and managerial maladies to be resolved completely before improving the education and experience of software developers. Fortunately, most software professionals are eager to hone their technical skills, so future impediments to successful reuse will be largely self-imposed.so now the companies look at horizontal reuse for efficient reuse.

5. REFERENCES

[1] Frederick P. Brooks, ``The Mythical Man-Month," Addison-Wesley, Reading, MA, 1975.

[2] Frederick P. Brooks, ``No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, Volume 20, Number 4, April 1987, 10-19.

[3] Brian Foote and Joseph Yoder, ``The Selfish Class," in Pattern Languages of Program Design 3, eds. Robert C. Martin, Dirk Riehle, and Frank Buschmann, Addison Wesley, 1997.

[4] Richard P. Gabriel, ``Patterns of Software, Tales from the Software Community," Oxford University Press, 1998.

[5] Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[6] Buschmann et al., Pattern-Oriented Software Architectures, Wiley & Sons, 1996.

[7] Douglas C. Schmidt, David Levine, and Sumedh Mungee, ``The Design and Performance of Real-Time Object Request Brokers," Computer Communications, Volume 21, No. 4, April, 1998.

[8] Douglas C. Schmidt and Mohamed Fayad, ``Object-Oriented Application Frameworks," Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.

[9] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley and Sons, 2000.

[10] Douglas C. Schmidt and Steve Huston, C++ Network Programming: Mastering Complexity with ACE and Patterns, Addison-Wesley, 2001.

[11] Douglas C. Schmidt and Adam Porter, Leveraging Open-Source Processes to Improve the Quality and Performance of Open-Source Software, 1st Workshop on Open Source Software Engineering, ICSE 23, Toronto, Canada, May 15, 2001.

[12] ML Griss and RR Kessler, Building Object Oriented Instrument Kits, Object Magazine, May 1996.

[13]R Malan and T Dicolen, Risk Management in an HP Reuse Project, Fusion Newsletter, April 1996

[14]W Frakes and S Isoda, Systematic Reuse, Special Issue IEEE Software, May 1994.

[15] J Hooper & R Chester, Software Reuse Guidelines and Methods, Plenum, 1991.

[16] EA Karlson, Software Reuse: A holistic approach, Wiley 1995.

[17]W Schäfer, R Prieto-díaz and M Matsumoto, Software Reusability, Ellis Horwood, 1994.