Developing Software for Large-Scale Reuse (PANEL)

Ed Seidewitz, *NASA Goddard Space Flight Center* (moderator)
Brad Balfour, *SofTech, Inc.*
Sam S. Adams, *Knowledge Systems Corporation*
David M. Wade, *Computer Sciences Corporation*
Brad Cox, *George Mason University*

1 Background

Software reuse is a simple idea: reduce the cost of software development by developing less new software. In practice, however, achieving software reuse on a large scale has been frustratingly difficult. Nevertheless, as the complexity and cost of software systems continues to rise, there has been growing interest in really trying to make large-scale reuse a reality. Interest in reuse has been particularly strong in both the Ada and object-oriented programming communities. This panel brings together key experts from these communities to discuss the issues, problems and potentials of reuse. This discussion is continued from the first meeting of this panel at the Tenth Washington Ada Symposium last June.

Brad Balfour and David Wade represent two major projects using the Ada programming language. Brad Balfour is involved in the DOD Corporate Information Management effort, which, as part of the DOD Reuse Initiative, is attempting to realize widespread consolidation of DOD MIS software. David Wade works on the FAA Advanced Automation System project, a very large, complex project which provides a rich domain for reuse.

The other two panelists are well known in the object-oriented programming community. Sam Adams is a recognized expert in object-oriented design and programming and has been working recently on new approaches to reuse for Knowledge Systems. Brad Cox is the creator of the Objective-C programming language and has had considerable experience with the development and marketing of reusable software components. Currently he is studying the economic approach required to truly achieve wide-spread software achieving reuse.

Each panelist has been asked to focus in their position papers on a specific area of the panel topic:

Brad Balfour	Language Issues
Sam Adams	Methodological Issues
David Wade	Practical Experience
Brad Cox	Economic Issues

While each panelist did, more or less, direct himself to the given focus, it is interesting to note that their seems to be a consensus that the most difficult problems in achieving reuse are, in the end, largely economic and managerial. While technical considerations, especially object-oriented methods, have an important place in enabling reuse, it is approaches to resolving the even more difficult non-technical issues that may provide the most interesting debate during this panel session.

2 Brad Balfour

Within both the reuse and development methodology communities, it is fashionable to claim that implementation is a nearly insignificant part of the overall life-cycle, and therefore, the choice of an implementation programming language is a non-issue. However, the language used to implement reusable software assets will, in fact, have a profound impact on almost all of the technical aspects of reuse. The choice of an implementation language will impact: (a) all supporting technologies used to accomplish reuse, (b) the economic conditions upon which reuse arguments are based, and (c) all "upstream" reusable assets including requirements specifications and design architectures. ***Some languages are similar and switching among them is simple, other languages are wildly diverse in the capabilities they support and choosing among them represents an early, fundamental choice.***

When implementing a library of reusable software assets, the primary impact of choosing a specific implementation language will be on the supporting reuse technologies including: developing reusable assets, certification of those assets, and using the assets. The *guidelines for developing software* assets which will be reusable are *not* language independent. While some guidelines can be made to sound language independent (e.g. "Separate interface from implementation"), they may not be realizable in all implementation languages — therefore belying their language dependent nature (e.g. in Eiffel and Ada interfaces are easy to separate from implementation, but how does one follow the principle in COBOL or FORTRAN?). The supporting technology of *asset certification* (the process of measuring assets for their reusability) is also language dependent. The definition of what makes a asset reusable differs depending upon the language used, and the definition's measurement is also language dependent. This makes certification a language dependent process. Similarly, the *process of reusing an asset* is language dependent. Some languages (e.g. Ada) use dependency and generic instantiation to incorporate parameterized components into a newly written program; other languages (e.g. C++) use inheritance and physical inclusion (copying via #include) to incorporate existing assets. While other languages (FORTRAN) have no mechanism other than physical copying of source code, modification of common blocks or source code bodies and calls to allow for assets to be reused.

The choice of an implementation language has a major economic impact upon the amount of reuse achieved. To maximize the return on one's investment in the creation of an asset, it is important that the asset's cost be amortized over as many users as possible. In order to create the largest audience for an asset, an organization is driven to standardize on the implementation language used for that asset. Otherwise, the market for an asset will be the much smaller subset who use the one language the asset is written in and will leave out users of other languages.[1] This motivation is what underlies the fact that many organizations which use UNIX based systems have standardized on C and that the DoD, as an organization, has standardized on Ada — it makes economic sense to have a single, common language to maximize the number of people who will reuse a given source code component.

The choice of a language deeply influences the form, content and reusability of all "upstream" products. Domain specific software architectures (designs) are both language and methodology dependent. Additionally, there is a strong link between the methodology chosen and the language used to implement the design. Once the design methodology is chosen, this choice influences the method used for requirements analysis and the domain model (e.g. an object-oriented design doesn't work well to support structured analysis). Languages can be group in levels according to their similarity: C++, Smalltalk, Ada9X vs. Modula-2, Ada 83; vs. COBOL, FORTRAN and C). Within each group, it is relatively easy to move from one language to another as they provide equivalent reuse support. However, it is difficult to move from one group of languages to another. As an example, while group one provides good support for an object-oriented architecture, a choice of an group three implementation language would make it difficult to reuse an object-oriented architecture.

There are several ways that the choice of an implementation language will affect the entire spectrum of reuse from Domain Model to Domain Specific Software Architecture to source code component. It is important to recognize that these

[1] It is technically possible to use a source code asset written in another language and either call on it (through an inter-language call) or to translate it into the language the client is using. However, both possibilities tend not to happen in practice, as teams try to avoid mixing languages within a single program.

are not just technical considerations, but also include important economic arguments for widespread standardization. Unless both the technology of mixed language development and the culture surrounding "languages as religion" change, the early choice of a programming language will continue to have a large impact on reusability.

3 Sam S. Adams

The Business of Reuse

Making reuse happen on a large scale places requirements on almost every aspect of software development. There are two main business goals that drive the need for large scale reuse. The first is the need to reduce the total "cost of ownership" of the organization's software. The second goal is the need to convert software from a long term financial liability into a corporate asset that continues to provide a high return on investment as it is reused in project after project.

Today, roughly 80% of the total cost of software occurs after initial delivery. Some of these costs arise from problem correction, but most are generated by changing requirements and requests for new or extended features. Unfortunately, the software industry has traditionally focused its efforts on optimizing the software creation process, which accounts for only 20% of the overall cost of ownership. Software reuse by its very nature requires a larger, multi-project view of development, where evolution, refinement and reuse of software assets is essential.

The Right Product

So, we need to be able to create and reuse these software assets on a large scale throughout our organizations. What kind of software is suitable for this level of reuse? First of all, the software must be "reuseful". Reuseful software embodies some behavior that is commonly needed in the applications developed by an organization. If the software isn't useful, then it doesn't matter how reusable it is, it's simply worthless as an asset. But assuming that the software will be useful in many kinds of applications, what else is required?

Searching the shelves, reading the labels

The developer must be able to locate the software, understand its behavior, and trust that it will deliver what it promises in a robust, high quality manner.

To make the software easy to find, a reuse library system will need to be deployed and software classified properly for easy location and retrieval.

Once a developer has located a software component, however, he must determine its behavior to see if it will serve his purposes. Module descriptions and programming code are simply insufficient mediums for this kind of information. A much richer medium that consistently manages all relevant analysis, design, implementation, testing, and usage information is required. Keeping all this information in sync with the executable software is essential for large scale reuse.

Quality and Trust

Once a developer has selected a software component for reuse, how can he determine the quality of the component? Reuse requires trust. If a poor quality component is reused in an otherwise high quality system, the overall quality of the system suffers. But how can you tell? Software suitable for large scale reuse must pass both rigorous testing and stringent metrics, with both the test results and metrics included in the library to assist in the decision making process. Also, the tests themselves must be available to test the component's behavior within the context of the application where it is being reused.

The Right Process

Now that we know some of the characteristics of the software we need to develop for large scale reuse, what kind of development process is required? If the software is to be reuseful as well as reusable, careful attention must be paid to the design of those components that model the essential nature of the business domain being addressed. This "domain modeling" calls for an iterative, people centered process driven by the insights of those individuals that have the greatest knowledge of the business domain, the users, their managers, and other domain experts. The process must be accessible to these people, for they will judge the ultimate success or failure of the systems being developed.

Constant Quality Management

The process must also be tuned to produce high quality software. This requires an iterative development process with short cycle times and high degrees of quality feedback. Long periods of development without testing and quality assurance provides to much opportunity for a project to lose sight of its quality goals. By shortening the time between a change and its evaluation, the project can never get too far off track. And while testing is an essential element in quality assurance, pass/fail testing can only produce "software that doesn't break". Creating software with "Quality that Fascinates" requires high frequency measurement of qualitative metrics that promote a process of continual quality improvement.

Reuse Creation and Utilization

While the focus of this panel is the development of new reusable software, the process used must explicitly encourage and expect reuse itself. Developing reusable software should not be a special activity, it should be a natural part of the entire development process. Whether a software component is reuseful or not, making software more reusable increases the quality of its design and implementation.

Staying in Sync

Finally the process must maintain the consistency of the requirements, analysis, design, implementation and testing information for the software component. If the code works but does not reflect the design or requirements, it has little value to anyone, especially as an asset to the organization.

Summary

The creation and reuse of high quality, reuseful software assets is essential to successful large scale reuse throughout an organization or throughout the industry. Object Technology has provided a foundation for these software assets, but more powerful mediums such as Well Defined Objects [1] will be required to maintain the information required for large scale reuse. But having the right medium does not ensure success. Adopting an integrated approach based on the iterative refinement and constant quality management of the complete software lifecycle will also be required [2].

References

[1] Adams, Sam, "Object Transition By Design", Object Magazine, November/December 1992

[2] Adams, Sam, "Return On Investment: Constant Quality Management", The Hotline on Object Technology, Volume 4, Number 1, November 1992

Software Assets and Well Defined Objects are trademarks of Knowledge Systems Corporation, Cary, North Carolina, USA

4 David M. Wade

The Federal Aviation Administration's (FAA) Advanced Automation System (AAS) is the largest of the FAA's projects to upgrade the nation's Air Traffic Control (ATC) system. This massive software development undertaking is being performed in accordance with DoD-STD-2167 by IBM (the prime contractor), Computer Sciences Corporation (CSC), and a variety of other subcontractors. Although CSC is responsible for developing over a million of the estimated 2.5 million lines of Ada code that will be required, the size of the effort is only one of many challenges. AAS is also complex, distributed, real-time, interactive, long-lived, and has high extensibility, availability and reliability requirements.

Modern Software Engineering concepts have always been stressed on AAS and were one of the reasons the Ada language was selected [1]. A high level of software reuse was considered an important facet of these concepts early on. Aside from the use of external software repositories and Commercially

Available Software (CAS), a strong effort was made early on to foster "internal" reuse between the highly independent development groups that make up AAS. In June 1989 a Reuse Working Group was formed to aggressively pursue this goal. Its original limited mandate was to study the existing reuse effort on the project and suggest improvements. It was quickly recognized that considerable software reuse benefit could be gained just through a sustained coordination effort, therefore the group was transformed into a permanent entity. Shortly thereafter the CSC Reuse Group (RG), a subset of the Reuse Working Group, was formed to actively support the AAS Reuse effort within CSC. The RG has since initiated a variety of innovative techniques to foster reuse and has even been allocated staff to develop a large percentage of the core Reuse software.

The RG devised a three pronged attack on the many obstacles to software reuse that are common on all large software efforts. First, there was a strong emphasis on education. From the beginning RG members have given presentations to the AAS community on the advantages of Object Oriented Development (OOD) techniques and software reuse. These talks were carefully focused to specific audiences so that hard, cost/benefit information could be provided to managers while developers would be shown the many technical advantages of software reuse. This effort was important in building the grass roots support that is necessary for a successful reuse effort and at the same time convinced management that it would result in a net gain for the bottom line. These talks have since been consolidated into an eight hour, two day training course on software reuse that is now mandatory for all CSC personnel new to AAS.

Second, the RG ensured that metrics were kept for important aspects of the effort. In order to justify the existence of a reuse effort it was known that tangible evidence of increased effective productivity would be required. This meant keeping track of what was being reused, who was using it and the effort required to develop it. The metrics that have been compiled have required only basic information to be maintained and surprisingly few automated tools.

The last key element of the RG effort has been its full participation in the development effort. Although originally conceived as a steering committee, it soon became apparent that the best way to convince people of the viability of software reuse was for the group to become involved in the hard, practical work of code development. The reuse development effort began with a pseudo domain analysis of Air Traffic Control systems which was used to create a taxonomy for software components. A process was then created to identify reuse components, allocate development work and track the development, maintenance and use of these components.

With staff available the RG was able to take on the responsibility of developing any software that didn't clearly fall under the control of one of the standard development groups. In addition, certain components were recognized as being critical to the success of the reuse effort and having staff on hand allowed the RG to make sure these components were developed in a manner consistent with the long term goals of software reuse.

Although the RG had very practical reasons for participating in the development effort there were several surprise benefits that were garnered. First, the RG's involvement at this level provided the group with invaluable insight to the problems (and opportunities) unique to the AAS development environment. Also, by participating with the AAS community in the day to day software development effort, the RG was able to engender additional support for its efforts.

Currently the reuse component library consists of over 60 components made up of more than 60,000 lines of Ada code of which more than half have already been developed and are in use by the AAS development community.

References

[1] Basili, Dr. V. R., Boehm, Dr. B. W., Clapp, J. A., Gaumer, D., Holden, Dr, MTR-87W77, April 1987.

5 Brad Cox

The software crisis is a quarter-century old; as old as software engineering itself. Isn't it time to recognize that this crisis is a symptom of a deeper problem than software engineering's traditional infatuation with programming languages and methodologies could ever get at? Isn't it time to stop and reflect on why software engineering is still delivering at best arithmetic improvement, while our colleagues intangible engineering domains make exponentially growing improvements seem almost routine?

Whereas hardware engineers speak of buying and selling hardware components in a *market*, we speak of *reusing* software components from a *repository*. So long as we neglect market processes and the revenue mechanisms that fuel them, we will remain a primitive tribal economy, quite unlike the mature engineering societies to which we aspire. Electronic repositories and networks will not be shopping malls. They'll be garbage dumps where those who don't mind sifting through other people's garbage will hunt for Good Stuff Free, wondering why quality-conscious engineers' response to such reuse is "Not in *my* backyard!"

We've paradoxically defined software engineering to mean the craftsmanship of primitive, unspecialized, tribal societies and arbitrarily exclude the kind of engineering most characteristic of advanced industrial societies. Software engineering is synonymous with programmers; those who fabricate components from first principles. The term generally excludes end-users; those who customarily build their own custom solutions out of shrink-wrapped pre-fabricated components.

For example, consider a financial analyst who assembles a desktop publishing engine by buying shrink-wrapped word processors and spreadsheets, and uses them to process a Dow Jones stock quotation data-feed in order to write financial articles for publication. Isn't this analyst as much engaged in advanced engineering as a refinery engineer who builds a refinery to process petroleum? Isn't a Smalltalk programmer who assemble classes from a class library doing engineering in an equally advanced sense of the term, but at several levels lower in a specialized labor hierarchy? And isn't a C++ programmer who builds applications from components that he fabricated solely for the project on hand engaged in a primitive kind of engineering, indistinguishable from an aborigine basket-weaver's hand-craftsmanship?

Of course, our paradoxical definition of "software engineering" is neither capricious nor malicious. It is a consequence of the fact that electronic products are so ephemeral that it has never been obvious how to treat them as "products" in the sense that ore, metals, and silicon chips are products that can be robustly bought and sold by the copy. The cause of the symptoms we call the software crisis does not originate in the software development process, but in the easy-to-copy nature of the products thus produced. Unlike the hard-to-copy tangible goods of the manufacturing age, information age goods can be copied in nanoseconds and transported at light speed. This undercuts the market processes upon which manufacturing age industries have achieved the very maturity to which we aspire.

The market mechanism for the tangible goods of the manufacturing age didn't require any particular attention. The hard-to-copy nature of tangible goods made the traditional pay-per-copy mechanism the natural choice. But the market mechanism is very much an issue for information age goods that can be copied in nanoseconds and transported at literally the speed of light. This so thoroughly undercuts the pay-per-copy mechanism of traditional markets that there is considerable dispute as to whether a robust supply of pre-fabricated information age goods is even possible.

The software crisis can be solved just as the problems of cottage industry manufacturing were solved during the industrial revolution [1]. Not by the accustomed nostrums of software engineering as we've defined this term in the past, but by enabling the same mechanism that drove manufacturing industries to its successes. I'm referring to the market processes that coordinate the self-interested activities of large numbers of individuals,