Managing Change to Reusable Software

David Kane (Applied Expertise, *dkane@aecorp.com*)

William Opdyke (Lucent Technologies/ Bell Labs, *wopdyke@lucent.com* OR *william.opdyke@bell-labs.com*)

David Dikel (Applied Expertise, *ddikel@aecorp.com*)

Applied Expertise	**Lucent Technologies** **Bell Labs Innovations**
1925 North Lynn Street Suite 802 Arlington, VA 22209 (703) 516-0911	2000 N. Naperville Road Naperville, IL 60566-7033 (630) 979-0730

Abstract:

Change is one of the few "constants" of software engineering. While managing this change is a challenge for all software-intensive organizations, managing change becomes more difficult when organizations build product-lines. While reuse helps manage change across the product-line more effectively, managing change for specific components becomes more difficult. This paper describes six organizational patterns that support software reuse and address these concerns.

Subject Area:

Software Reuse

Keywords:

Reuse, Management, Organization

Introduction:

Change is one of the few "constants" of software engineering. When organizations undertake software reuse, the need to manage change for components grows due to the expanded usage and often an expanded life-cycle of a software assets. These changes need to managed in a coordinated fashion [*FooteOpdykePLoP94*].

This paper describes six patterns that support changes to reusable software:

- *Develop a Shared Platform*

- *Maintain Reuse Platform Identity*
- *Integrate Reuse and Tie to the Bottom Line*
- *Reuse More than Just Code*
- *Treat Reusable Components like Products*
- *Merge After Cloning*

Some of these patterns were mined during the recent workshop on software reuse, where a working group investigated organizational issues, focusing on practices that support managing the rhythms of change and managing cloning to maintain a shared code base [*WISRReport97*]. The workshop included participants from leading industry organizations such as AT&T, Lexis-Nexis, Lucent, and Motorola. In defining this set of patterns, the authors have also drawn upon their experiences. Bill Opdyke has been conducting several platform and reuse related projects at Lucent Technologies, consulting with platform and product development groups. Bill's research into refactoring patterns to support evolution and reuse is described in a paper coauthored with Brian Foote and reviewed at PLoP '94 [*FooteOpdykePLoP94*]. Dave Dikel and Dave Kane have conducted a number of studies and benchmarks in the area, including the 1996 Software Reuse Benchmark (SRB). SRB included 11 industry and government organizations, including Andersen Co nsulting, AT&T, Computer Sciences Corporation, HP, and Texas Instruments. [*WilsonDikelSRB96*]

Pattern #1: Develop a Shared Platform

Problem Statement

Multiple projects incur redundant development costs by independently implementing identical or similar functionality.

Context

Projects within an organization are loosely-coupled, allowing each project to more effectively focus upon meeting the needs of its set of customers. However opportunities might be realized to achieve economies of scale by centralizing some functionality in a common platform.

Forces

- Multiple implementations of similar functionality are often more expensive than a shared solution.
- Projects want to minimize their development costs.
- Projects want to maximize the revenue potential for their products and services.
- The existing organizational structure may not easily support coordination across projects.
- The needs among projects, while similar, may not be identical - or at least may not appear to be identical to the project staff.
- Expertise in common areas may be spread across projects; such staff may be in high demand within their projects.

Solution

- Bring together staff representing multiple projects, to assess common needs.
- If the areas of common need are significant, charter the development of a shared platform, funded and staffed either jointly among the application projects, or by a corporate level "core"

organization.
- Define plans for projects to transition onto the common platform.
- Maintain close ties between the platform projects and the application projects, to both help focus the platform development and increase the confidence level among application projects that the platform will meet their needs.

Result

A platform is developed to address the needs shared among the application projects. Cost savings (in areas addressed by the shared platform) are realized among the projects, over time. Time-to-market is also reduced. Subsequently, opportunities may also be realized to share application level components among projects. Opportunities to migrate staff among projects may be increased given their shared platform understanding.

Consequences

For the benefits of a shared platform to be realized over time, it is important to *Maintain Reuse Platform Identity.* An investment is required to establish the platform before the benefits can be realized.

Example

In a government organization that builds combat and training systems, 28 different software products on 128 different ships were maintained separately. Maintenance changes took years to implement. As budgets became tighter, a new approach was needed to continue to carry out the mission and ensure the survival of the organization. Now that the organization has adopted a shared platform across these products, costs are substantially lower - 5% to 21% of the costs without such reuse based upon Gaffney/Durek calculations. Time-to-market and quality have improved as well [*WilsonDikelSRB96*].

Related Patterns

- *Maintain Reuse Platform Identity*

Pattern #2: Maintain Reuse Platform Identity

Problem Statement

A shared platform has been built, or is envisioned, but potential users, aside from the pilot adopters, are not willing to commit to using the shared platform.

Context

Over time, user needs change, technologies and interfaces change, and designers' insights change [*OpdykeRobertsOOPSLA96*]. Change is one of the inherent complexities of software [*Broooks86*]. Platforms and frameworks have been recognized as a means for managing change and for reducing development costs and development intervals. As a result the organization has decided to *Develop a Shared Platform*.

New platforms are often introduced with a pilot adopter. While it is necessary for the platform and pilot

application development groups work closely together to make the initial use of the platform successful, it is tempting to associate a platform project too closely with the pilot application. This association is sometimes realized by placing platform staff and application development staff in a common organization. This close association may benefit the application developers with access to and control over the platform development staff.

From the application developers' perspective, distinctions between platform and application are irrelevant to their customers. It may seem expedient to make arbitrary changes to the platform to accommodate application specific needs, in ways that compromise the generality of the platform. A further expedient may be to merge application and development staff; in the process, platform identity and platform support may be lost.

This close collaboration can cause other potential adopters to see the platform as only suitable for the pilot application, or that the pilot application has too much control. As a result of such perceptions, other applications groups don't commit to using the platform. A warning sign that the platform identify has not been maintained is that the schedules for platform adoption after the pilot application begin to slip.

Forces

- Costs can be reduced and economies of scale achieved through the use of common platforms.
- Change is inevitable.
    - ironically, introducing change is hard.
    - pilot projects mitigate some of the challenges of introducing new technology.
- Platform development groups are driven by multiple, conflicting goals
    - near-term focus of the application development groups using the platform
    - long term requirements for potential adopters of the platform.
- Application development groups are focused on near-term deliverables to specific customers and markets. In order to maximize product quality while minimizing product development intervals and development costs, application development groups seek to reduce and eliminate:
    - unnecessary dependencies between organizations
    - unnecessary distractions placed upon their staff.
- Funding sources often drive the priorities of an organization.
- Organizations that are closely tied together (e.g. in the same business unit) often have better communication than organizations that are not as close.
- Potential platform users that are not confident that a platform will be responsive to their needs will not use the platform.
- A platform closely tied to a single project may not be seen as responsive to the needs of other users.

Solution

- Establish a separate organization for the platform.
- Keep the staff of the platform development organization and the initial application organization separate. There should be good lines of communication, and clear, commonly understood expectations between the two groups.
- Use stable corporate funding to get the platform started, and then ensure stable funding from the application organizations once the platform is established.
- There needs to be an advocate for the platform at the corporate level.
- The platform organization should act like a product organization to engage customers and address

customer needs.

Result

If platform identity is maintained, applications are better able to have their needs met, and the value of the platform is greater because it can be used by more applications. If the platform identity is maintained, the the platform will be used more widely, which results in reductions in cost and time to market across a larger portion of the organization.

Consequences

Maintaining platform identify may make achieving success with the initial application more challenging.

Related Patterns

- *Develop a Shared Platform*

Pattern #3: Integrate Reuse and Tie to the Bottom Line

Problem Statement

Incentives to establish reuse are not taken seriously [*WilsonDikelSRB96*].

Context

The organization has multiple improvement efforts going on simultaneously, e.g. process improvement, new tools, risk management - as well as an organization wide reuse effort. The reuse effort is being ignored and rejected by the organization as just "one more" activity to distract engineers and managers from meeting the business objectives.

Forces

- Improvement efforts are related - but staff focusing on a particular improvement effort may be unaware of other, related improvement efforts.
- Organizations are often under tremendous pressure to meet business objectives of completing projects on time and on budget.
- Some level of reuse often happens, even without a formal reuse program.
- Organizations cannot be motivated by mandate.

Solution

- Reuse leaders should identify fit and ways to serve and be served by other improvement efforts.
- Coordinate and align the reuse effort with other improvement efforts. The organizational structure should support shared responsibility among the improvement efforts.
- Identify and communicate common, coordinated recommendations to practitioners regarding the improvements - emphasizing the value of the improvement efforts to the managers and engineers whose support and actions are needed to make the improvement successful.
- Manage the coordination among the efforts.

- Track, on an ongoing basis (and require estimates in budget proposals for) how much reuse has been achieved and how much savings have been realized.
- Integrate reuse throughout the life-cycle, not just at coding.

Result

The engineers and managers involved with the reuse activity receive consistent, easy-to-understand and follow improvement guidance that shows them how to better achieve their business objectives. The reuse effort is more likely to take hold, and deliver the expected results of reduced costs and reduced time-to-market.

Consequences

Even if reuse is tied to other improvement efforts, the improvement efforts still need to be tied to the bottom-line and other motivations of both managers and engineers. If the value of the improvements to the participants is not clearly communicated, the integrated improvement efforts may be more likely to fail. Even with careful coordination and consolidated "improvement" strategy, the effort may be seen as more complex than each of the efforts individually. The reuse effort could also be canceled if other improvement efforts are deemed more critical (this may not be bad).

Example

In one organization, managers and practitioners believed that there were too many "initiatives." To implement reuse, the organization realized that reuse must be integrated with existing software and process improvement efforts. The software engineering process group defined, managed, and maintained the software processes, and they incorporated reuse into these processes. Before this shift to integrate improvement efforts, managers received a confusing array of recommendations. Afterward, they received an integrated set of processes that included reuse from the software engineering process group. Because reuse has been integrated into the development process, it gained widespread cultural acceptance within the organization. The organization improved their planning and decision making, reduced cycle-time, and won new contracts.

Related Patterns

- *Develop a Shared Platform,*
- *Reuse More Than Just Code*

Pattern #4: Reuse More than Just Code

Problem Statement

Engineers don't reuse already developed code components because the components either don't fit with other components, or do not quite fit the requirements [*WilsonDikelSRB96*].

Context

Managers and practitioners are beginning to make a dedicated, long-term effort to improve results and customer value. Software reuse is seen as a way to achieve these results.

Forces

- Managers and practitioners often equate reuse with code, not with knowledge capture or organizational learning [*Simos*].
- If different parts of the organization have very different practices, then they may find it more difficult to share assets and share people among different groups.
- Code components reflect decisions made during requirements and design.
- Consistently achieving high levels of process maturity requires an understanding and application of sound requirements and design techniques.

Solution

- Reuse software assets from all phases of the life-cycle including:
  - Processes
  - Checklists
  - Project management templates
  - Requirements
  - Designs.
- Establish a repository and other automated tools to share the assets.
- Tie reuse to knowledge capture activities.

Result

A number of benefits have been attributed to this approach, although not all organizations realize all of the benefits. Creating and reusing non-code assets makes code reuse more likely. Organizations are also better able to bring in and train new staff. Reusing non-code assets also results in more direct benefits such as improved cycle-time, consistency and repeatability and reduced costs, risks and surprises.

Example

One business unit established an online repository. The repository included documentation of the current processes, project information, and reusable templates, plans, and constraints. Because different groups used the same processes, it was easier to move employees between groups, and it was easier to bring new people up to speed. The business unit was able to grow while at the same time reducing documentation costs by 40%.

Related Patterns

- *Integrate Reuse and Tie to the Bottom Line*

Pattern #5: Treat Reusable Components like Products

Problem Statement

Application developers typically associate components with "use at your own risk stuff" -- they don't trust or use components they don't fully understand.

Context

An internal organization is chartered to introduce a new set of behaviors and/or a new technology through their development and management of reusable components. The goal of this activity is to achieve a significant improvement in delivering higher value and better business results.

The target adopters are application development organizations. The application development organization is under tight deadlines. Those who supply components must support a diverse group of product developers, each with his or her own needs. Even though each component has one owner, others may need to make changes to his or her component when he or she is away to meet deadlines.

A warning sign that this problem is occurring is that component owners do not hear about change requests from component users, either because components are not being used at all, or because a configuration management (CM) system is not being utilized to capture and communicate change requests.

Forces

- If individual developers are not confident that component features critical to their work will not change, and that new component features will be addressed in a timely fashion, then developers will probably not use the components.
- The supplying organization must must add new features and feature sets to its components to increase its customer base.
- Potential adopters will not use a component if they are at all unsure that it will be maintained and that it will be managed -- unexpected defects coming from lack of CM are *not* cool.
- With a standard configuration management system, only one component owner has access to a component.
- A standard configuration management approach focuses on tracking baselines of entire systems, e.g. NamedStableBases [*Coplien97*], and not tracking specific components across multiple systems.
- Without a product mentality, component suppliers can easily become detached from their customers
- Lack of information raises the perceived risk of using a component. Perceived risk on the part of (potential) reusers causes them to lose confidence that a component will meet their needs, and hence makes them reluctant to reuse the component.
- Potential reusers are often willing to forego their desire to know everything about the asset's internals and if they trust the supplier's credentials and know that the supplier has a competent CM tool and process.

Solution

Treat reusable components like products. A lot is known about how to build and market products that go within other products, (though reuse efforts still do believe they are doing something entirely different). For example, the organization responsible for reuse should gain the confidence of the target customer base through building a high-quality product, tracking and responding to change requests. Gain endorsement from a respected expert and end user, and draw to your team a group of developers that people trust. The reuse organization shares developer's risk through specific commitments to deliver the features promised, managing and responding to change requests, and assuring that critical features will persist. The benchmark is set by product vendors (who certainly are not perfect).

Make use of user groups and other communications forums to encourage the platform user community to exchange information and drive consensus regarding proposed platform changes.

A flexible CM system and process is a part, but not all of the solution. Developers should be able to view version histories, and outstanding change requests. Users should also be able to use the CM to report defects and make other change requests.

Result

Reusable components are presented in a way that developers are more comfortable with. They are willing to try using components because they have a clearer understanding of who is responsible for what. They see the risk of using a component counteracted with endorsements from their peers and experts they respect.

Use of reusable components grows from small groups within product organizations to a point where major product features may depend on the components.

User community is better informed about specific reusable components. This knowledge reduces their perceived risk to reusing components, and increases component usage. The flow of information helps develop relationships between the component owners and reusers.

Consequence

As usage grows, a problem in a component may adversely affect multiple business-critical products.

Example

In the early days of acceptance of C++ in the AT&T development community, it was determined that acceptance would go faster if users were supplied with a set of standard "computer science" functions that would make system development using C++ easier. This set came be known as the C++ Standard Components Library and contained functions like: Lists, Maps (associative arrays), String, Blocks (unbounded arrays with memory allocation as needed) and so on. The Standards Component Library was very popular and was used by hundreds of projects across AT&T. The components were put under configuration management control and change requests (bug reports and enhancement requests) were tracked in the configuration management system, which associated each source change with specific change requests. Any developer assigned a change request could change a component and an audit trail of the change would be kept by the system.

The C++ hotline used the configuration management system to enter customer change requests, track the quality and change history of the components and support the assembly and packaging of new releases. The project's configuration management was more than just the supporting system, it included the process (e.g., review boards) as well.

Acceptance was high and cloning was not a problem.

The success of the Standard Components Library was based on many factors, which helped build confidence among potential users:

- Quality of people, many of whom published books and had excellent reputations within the company.
- Strong support from Bjarne Stroustrup
- The quality of the components and the support that was given. Configuration Management played a supporting role here in that without it, quality and support wouldn't have been as good as it was.

Releases of reusable components were not that much different from releases of other products that have to run in multiple environments.[*Cichinski97*]

Related Patterns

NamedStableBases [*Coplien97*]

Pattern #6: Merge After Cloning

Problem Statement

Organizations clone software to add components to quickly respond to customer requests, but the organization does not want to get saddled with the long-term costs associated with cloning.

Context

While cloning offers a means to quickly respond to pressure to develop new features quickly, cloning often has far-reaching consequences. When a developer clones, code is duplicated, complicating product tracking and management. It doesn't take too long before developers build new and complex extensions on the clones making them incompatible with the code base. As a result, code grows rapidly in size and complexity. This dramatically increased the maintenance burden for each product [*DikelKaneComputer97.*]

Periodic merging of clones is needed.

Forces

- Rapid response requires that developers have access to and ability to modify existing source code.
- It is often easier to change source by "copy and modify" (ie by creating clones) rather than by extending or generalizing the original source.
- White box access to source is necessary to respond rapidly to multiple quick response requests from end-customers.
- Generalizing existing components may require coordination not just with the component supplier, but also other users of that component.

Solution

- Merge the clones regularly.
- Provide application developers with "white box" access to at least selected parts of the platform source.
- When cloning, encourage developers to duplicate the smallest portion possible.
- Establish very clear rules for what types of changes can be made through cloning.

- Identify clones by watching for signs, such as duplicate functionality, email postings, etc, and then examine the code to determine if cloning is taking place.
- Allow only one clone of any given component at the same time.
- Encourage users (applicationdevelopers) to submit bug reports, bug fixes and suggestions for extensions and generalizations.
- Provide support to component users so that they can more easily use the generalized software.

Result

- Code base improves in robustness and reliability; bugs get fixed and stay fixed. There is a single point of maintenance, which simplifies the task of building products.
- Reduced maintenance costs. Every line of shared code in use, means that there are 2 or 3 lines of avoided code somewhere else.
- Application developers can respond to market pressures more quickly than if generalized components were built of modified initially.
- Reuse suppliers build trust with users
- Code base stays upward compatible.

Consequences

Additional effort must be spent to reincorporate cloned software. There is also a substantial risk that cloning could get out of hand, e.g. cloned chunks are too big, or there are are too many clones, and making the practice too expensive to sustain. There may be limits to the number of simultaneous clones that an organization can create and still merge them back into the base. An organization may have reached or passed this limit when the time required to merge the clones slows down, or if a merge cannot be completed before the next scheduled merge. Another warning sign is that merges cannot be completed with major changes.

Example

One organization, in order to meet schedule, cloned their product in order to run the application in an additional operating environment. The organization recognized that in doing so, they had effectively doubled their maintenance costs, and that there was no way that they would be able to support all of the operating environments in their plans. They moved back to a common code-base and reduced their cycle-time from 12-24 months to 12-14 months, and they also reduced their maintenance costs. [*WilsonDikelSRB96*]

Related Patterns

- Consolidate the Program to Support Evolution and Reuse [*FooteOpdykePLoP94*].
- *Support Reuse via Flexible Configuration Management*

Discussion

Our work in documenting software reuse principles and related practices has been submitted to the IEEE Reuse Steering Committee, which is chartered to define principles for software reuse. We are interested in mining additional patterns in this area and in relating our efforts to other reuse & organizational related patterns.

References:

[Broooks86] Fred Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering". Information Processing 1986 - Proceedings of the IFIP Tenth World Computing Conference (H.-L. Kugler, ed.), Elsevier.

[Cichinski97] Conversation and email with Steve Cichinski of AT&T Labs. Steve was responsible for the group that developed and evolved a CM tool in AT&T. He was in the same organization as the group that developed the C++ standard components. He was also a participant in the WISR organizational principles working group.

[Coplien97] J. Coplien, Organizational Patterns Web Site, viewed June 5th, 1997, *http://www.bell-labs.co m/cgi-user/OrgPatterns*.

[DikelKaneComputer97] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, "Applying Product-Line Architecture," *Computer*, v 30 n 8, August 1997, pp 49-55.

[FooteOpdykePLoP94] Brian Foote & William Opdyke, Lifecycle and Refactoring Patterns That Support Evolution and Reuse. Presented at PLoP 94; included in "Pattern Languages of Program Design" (J. Coplien, & D. Schmidt, eds.), Addison-Wesley, 1995, pp 239-257.

[OpdykeRobertsOOPSLA96] William Opdyke & Don Roberts, "Refactoring Object-Oriented Software to Support Evolution and Reuse". Tutorial presented at OOPSLA '96: 11th Annual Conference on Object Oriented Program Systems, Languages and Applications, San Jose, California, Octboer, 1996.

[Simos96] Simos, Mark, "Learning and Inquiry Based Reuse Adoption (LIBRA): A Field Guide to Reuse Adoption through Organizational Learning," Software Technology for Adaptable, Reliable Systems (STARS), STARS-PA33-AG01, February 6th, 1996.

[WilsonDikelSRB96] J. Wilson, D. Dikel, D. Kane, M. Carlyn, C. Terry, E. Cavanaugh, D. Johnson, "Software Reuse Benchmarking Study: Learning from Industry and Government Leaders, " sponsored by DISA Software Reuse Initiative, January 1996.

[WISRReport97] Report on WISR '97: Eighth Annual Workshop on Software Reuse, Columbus, Ohio, March 1997. in the September 1997 issue of ACM Software Engineering Notes.