Making Frameworks Work: a Project Retrospective

Marwan Abi-Antoun

Institute for Software Research
School of Computer Science
Carnegie Mellon University
marwan.abi-antoun@cs.cmu.edu

Abstract

Various issues make framework development harder than regular development. Building product lines and frameworks requires increased coordination and communication between stakeholders and across the organization.

The difficulty of building the right abstractions ranges from understanding the domain models, selecting and evaluating the framework architecture, to designing the right interfaces, and adds to the complexity of a framework project.

Categories and Subject Descriptors   D.2.13 [*Software Engineering*]: Reusable Software—Reuse models

General Terms   Design, Documentation, Management

Keywords   experience report, product lines, object-oriented application frameworks

1.   Introduction

For the past few years, software product lines and frameworks have been popular both in research and in the industry. The Software Engineering Institute (SEI) Framework for Software Product Line Practice [34] discusses nearly thirty key practices which encompass all of software engineering. The SEI has also documented a handful of case studies of successful product line projects. However, few case studies discuss *enterprise application frameworks*, that address specific application domains and support the development of end-user applications and products directly [20, p. 10].

The author recently spent three years in the industry working as a developer on a software product line project with an enterprise object-oriented framework as its platform: more than two years as a framework developer, and then almost a year as an application developer on one of the first commercial applications of the product family.

The goal of this experience report is to document some of the lessons learned by focusing on the issues that seem to make framework development harder than conventional application development. Section 2 gives an overview of the project. Section 3 discusses how building product lines and frameworks involves a larger degree of coordination between many stakeholders and across the organization. Section 4 discusses the difficulties of finding the right abstractions, including building the right domain models, creating the right architectures and designing the right interfaces. Section 5 concludes with some thoughts on how the project turned out and a summary of the top 10 lessons learned.

2.   Project Overview

The product line project will be referred to as Project BLUE and the framework itself as the BLUE Framework. Project BLUE was conducted in the software organization of a large multinatinal Company that is not primarily a software organization. As part of providing non-software services to its clients, the Company develops and maintains several software applications.

The applications spawn multiple sub-domains, each with different characteristics. The applications in one sub-domain handle very large datasets, perform complex numerical operations and use advanced two- or three-dimensional visualization. In another sub-domain, the applications manage data in a shared repository, capture data interactively and generate reports on the data. Most of these applications are interactive and provide elaborate user interfaces.

Some of these applications were more than a decade old and consisted of several millions of lines of legacy code. But they still provided an important source of revenue. Many of the legacy applications had become hard to change and evolve to meet the new and constantly changing business requirements [7]. In some cases, each major release cycle took several years. In addition, most of these applications were integrated only at the level of a common data repository. Multi-disciplinary teams often needed to collaborate around several of these applications in a given workday. But the closed, monolithic nature and the poor integration of the applications did not meet the users' needs.

Some of the factors contributing to the lack of integration between the legacy applications include:

- **Independent development:** applications were developed in geographically distributed centers with little collaboration between developers and forethought for integration. In fact, many of these applications were added to the Company's software portfolio through mergers and acquisitions and evolved mostly independently, even after they were acquired;

- **"Not Invented Here":** there was no policy to encourage reuse. As a result, there was little component sharing between applications. For instance, there were multiple two-dimensional (2D) and three-dimensional (3D) visualization components based on third-party libraries or custom in-house implementations;

- **Stovepipe mentality:** the development organization suffered from a stovepipe mentality. For instance, there was a separate data management group who designed and maintained the shared application data model — even though almost everyone in the organization complained about how unwieldy that data model had become.

Nimble competitors were both gaining market share and winning awards in performance and user-friendliness. So the Company had to deliver better integrated software to compete effectively in the marketplace. Senior management decided to build a software product line [10] for the next generation of applications to slowly replace the legacy system. Indeed, an examination of the legacy applications confirms the need for such a product line. One count placed the number of separate applications at over 150, although they widely varied in size, and some were being discontinued. There were no technical justifications that applications using 3D visualization did not share a common 3D component. In a few cases, different user classes with varying skill levels justified some differences across applications.

The technical leadership decided that the platform of the product line would consist of an enterprise object-oriented application framework developed on a third-party technology framework. A framework is defined as "a reusable, semi-complete application that can be specialized to produce custom applications" [20]. So each instantiation of the framework into an application would be a product in the product line. With the product line, the Company aimed to achieve the following advantages:

- **Better integration:** using a framework could ensure a tighter integration of the various applications of the product line, an important business goal;

- **Consistent look-and-feel:** better integration could achieve a consistent look-and-feel across the different applications of the product line. In turn, this would improve the usability and competitiveness of the application suite;

- **Extensibility:** a framework could be specialized in multiple ways to provide both extensibility and variability.

This could improve the openness of the application suite and satisfy the needs of outside clients;

- **Reuse:** frameworks provide coarse-grained reuse, i.e., reuse of analysis, architecture, design and code, as opposed to techniques that mainly promote code reuse. This in turn promised to reduce both development and maintenance costs.

Project Timeline. The Company initiated Project BLUE by having a core Framework Team develop the platform of the product line, i.e., the BLUE Framework. The Framework Team was to deliver a prototype implementation of the BLUE Framework and a limited internal beta release a few months later. At that point, pilot projects were conducted to gather valuable feedback for future development. Shortly after the pilots, Project BLUE was distributed into several sub-projects at several locations. The core Framework Team from the initial project was still responsible for developing the core framework services and had technical oversight over the other sub-projects. A beta candidate of the core framework was released a few months later. However, this version suffered from poorly integrated subsystems. A few more months were spent in rework, refactoring and additional features. At that point, a stable version was released to internal customers. Shortly afterwards, the Framework Team was re-organized and relocated to address integration challenges arising from a strategic acquisition. The plan was to release several applications on early versions of the BLUE Framework. But the first application making a limited use of the framework shipped much later than anticipated. Since the project followed the Rational Unified Process [26], at least nominally, we refer to its Inception, Elaboration, Construction and Transition phases.

The Author's Involvement. The author joined Project BLUE as a framework developer as the project was entering its Construction stage and remained on the Framework Team until it was relocated. He then joined an application group developing one of the first commercial applications of the product family until the application entered its testing phase. As an application developer, the author evaluated many of the decisions that were made in designing and implementing the framework, including some of his own. Working in these two roles within the same overall project and organization also gave him the perspective to evaluate closely the differences between application and framework development.

3. Coordination Challenges

Framework projects seem harder than regular development projects because they typically involve more stakeholders than application development projects. An application development project has developers and end-users. But a framework project distinguishes between framework developers, application developers using the framework to build applications and end-users, among other stakeholders. As a result, a framework project requires more coordination be-

tween the various stakeholders to achieve a shared vision. Furthermore, building a product line is a medium-to-long term goal. In this case, the project was distributed, so it required sustained coordination across time and space.

Stakeholder Coordination. Project BLUE was launched and kept under wraps for various political reasons while it was going through its crucial Inception and Elaboration phases. As a result, some key stakeholders may have been excluded initially. During the protracted Inception phase, the Framework Team developed various domain models and produced voluminous vision, requirements, architecture and planning documents mandated by the Rational Unified Process — except that these had received little external scrutiny.

Management then unveiled Project BLUE and held a large review with many important stakeholders represented. The reviewers expressed suspicion about the whole project, raised many issues, complained that they did not understand many of the abstract domain models and requested checklists of the features and attributes instead of use case models. In retrospect, the project should have validated the vision and the requirements gathered in Inception before proceeding to Elaboration. These big bang reviews held at the end of Elaboration were too little, too late [39, pp. 159–173]. At that point, too much effort had already been invested in the work products under review. The discussions during the review meetings revisited decisions made long ago, or questioned the motivation behind some of the proposed features. More frequent or informal face-to-face communication between the stakeholders may have clarified many of these misunderstandings much earlier, without generating massive documents that had to be reworked.

Proof by Demonstration. Management was concerned by the initial reception of Project BLUE and communicated that to the project leadership. The leadership of the core Framework Team then decided to stop building abstract domain models and focus instead on developing a prototype of the core framework functionality.

Proof by demonstration is "a ruthless focus on the development of a system that provides a well-understood collection of essential minimal characteristics" [33]. This can be a useful strategy to deal with important process risks such as adversarial stakeholders, requirements creep, analysis paralysis and overemphasis on artifacts. This strategy can also help address various organizational issues. First, stakeholders change during the development cycle — the lack of signoff of earlier artifacts prompts concern on the part of the new stakeholders. Second, stakeholders are not generally comfortable judging the project status by only reviewing artifacts. Often a demonstration is the only "proof" that matters. Finally, deciding on the contents of a formal demonstration can help prioritize the features.

One decision was to build a test application to demonstrate a typical instantiation of the framework. This test application was generalized later into a shared application con-tainer. Using "proof by demonstration" for several months helped Project BLUE gain momentum and manage its stakeholders' expectations better, as follows:

- **Management:** In many ways, the livelihood of Project BLUE depended on giving successful demonstrations to management, to convince them of the feasibility of the approach. In particular, a series of pilot projects discussed later culminated with demonstration-based reviews that impressed all the attending management representatives. "It rotates, it spins", proclaimed the VP of Product Development.

  However, when dealing with management, using proof by demonstration can be a double-edged sword. A few months after the successful pilot demos, the same VP of Product Development paid a surprise visit to the Framework Team. The team was busy laying solid foundations and making progress toward an intermediate release. So the team demonstrated some of the early functionality implemented with a more sophisticated architecture under the hood. The VP of Product Development seemed disappointed to see only a subset of the pilot demonstration features. "Two months ago", he exclaimed, "you showed me something that rotated and spun, and now you show me this!"

  The Framework Team learned to not let a great demonstration set unrealistic expectations or mislead the stakeholders with respect to the current status of the system. Some of these early demonstrations were more than "judicious fireworks [to] excite the customer but not give unrealistic expectations. Dazzle the customer with just enough spectacular functionality to leave them wanting more. Don't demonstrate extras that will not appear in the end product" [15].

- **Application Developers:** application developers evaluated early versions of the framework during pilot projects. They demonstrated using prototypes what parts of the framework met their needs, and what parts did not. This was more useful than arguing about framework features in endless meetings.

- **Product Champions:** proof by demonstration enabled the Framework Team to achieve consensus between the Product Champions. Months of requirements negotiations with the Product Champions, often using paper-based prototypes, showed little progress. So the Framework Team developed several high-fidelity prototypes, presented them to the Product Champions and incorporated their feedback into the following versions. This was done for several weeks until the Product Champions agreed on the initial set of features. Low-fidelity prototypes seemed less effective than high-fidelity ones, perhaps because the former required the Product Champions to think about the problem in more abstract terms.

Following the proof by demonstration approach did have its limitations however. During the early stages, progress

was judged mostly by the content of the demonstrations and not by rigorous metrics such the percentage of implemented use cases. Furthermore, a baseline that made it through a demonstration did not necessarily have an acceptable quality level since the demonstrations were heavily scripted to avoid any known problems.

Coordinating Development. Project BLUE teams used various coping strategies to partially alleviate some of the coordination problems and they seemed to work:

- **Coordination Meetings:** Initially, the various teams contributing to the framework and the various product groups did not communicate regularly. At one point, weekly coordination meetings were established. The meetings mainly involved project managers. Architects or developers participated if there were technical issues that demanded their attention. Although these meetings were often contentious, they got the various framework teams and the application teams to communicate regularly. In retrospect, it would have helped to establish these meetings earlier in the project.

- **Configuration Management Notifications:** Each check-in into the configuration management system, whether the artifact was an updated requirements document or modified code, generated email notifications. These notifications often included a detailed description of the changes, cross-references to change requests in the defect tracking system, as well as instructions for testing. Developers often expressed their design intent in detailed revision letters. These notifications were a simple but effective tool in communicating shared knowledge and coordinating the distributed development. Unfortunately, many configuration management systems do not have the notification feature enabled by default. In some cases, a developer has to force explicitly the generation of these notifications.

- **Defect Tracking System:** The process required using a defect tracking system to capture bugs and feature requests for the framework. Unfortunately, during some iterations, the team members found it often more expedient to exchange emails with reports of the various defects or enhancements. In retrospect, using the defect tracking system rigorously — as in the Eclipse open-source framework project, could have prevented forgetting about important items. Most application groups in the Company used the set of open issues in the defect tracking system as the main basis for iteration planning.

- **Developer Training:** Learning a framework is harder than learning a class library because you can't learn just one class at a time. Project BLUE scheduled framework training days for application developers at reasonably regular intervals, with each training event spread over several days. Most of the developers attending these sessions were assigned to projects that were contemplating the immediate use of the framework in their applications.

Each event was also a forcing function to bring all the documentation, samples and tools up to date. In a few cases, rerunning the regression tests before the training events found bugs that had been recently introduced. The training sessions also helped application developers network and share knowledge.

On the downside, some premature training sessions introduced the application developers to a version of the framework that would be obsolete by the time they started development on their applications.

Managing the Vision. Having a shared vision for any project is important, and even more so for a project building a product line. The project achieved this shared vision using an organization involving different *product champions*. A product champion played a role similar to that of a *program manager* in some software organizations. The main difference however was that product champions often had considerable domain expertise, advanced degrees in the sciences, but typically less formal training in software engineering or computer science.

- **Product Line Champion**: a Product Line Champion was assigned to the product line project. He authored several of the defining documents of the product line, including a comprehensive vision document.

- **Domain Champions:** the Product Line Champion relied in turn on a number of champions who were domain experts in their respective sub-domains.

- **Framework Champion:** a champion acting as the surrogate framework user was assigned to the core Framework Team. Targeting several domains often left the Framework Team with conflicting requirements or priorities. These contradictory requirements were often resolved between the Framework Champion, Domain Champions and the Product Line Champion.

- **Product Champions:** at a later point, two project managers from the first application groups targeting the framework were added as Product Champions. They helped set the priorities of the Framework Team, improved the coordination between the framework and those application groups, and made for a win-win situation. This was done late in the project since early framework adopters had not been designated.

Generally, software practitioners recommend avoiding intermediaries such as product champions as they can be considered "people, who, presuming to translate between real users and systems people, end up being noise in the channel" [3]. For a framework project however, intermediaries may be needed because there are more stakeholders. Furthermore, each instantiation of the framework may have a different set of stakeholders. In this case, the Domain Champions helped the Framework Team derive framework use cases from the application use cases in their sub-domains of expertise. The multi-disciplinary team of Product Champions from multiple domains provided the Framework Team

with the multiple application perspective that is crucial for framework development. In addition, several Framework Team members had built several applications. So there was no need to build several applications and generalize them into a framework, as is commonly recommended [31].

Maintaining the Vision. The champion organization worked reasonably well for Project BLUE to maintain the vision in the face of many changes. At one point during the project, there was a reshuffling in the champion organization. First, the Product Line Champion moved to a different position. This led to a period of uncertainty before his successor took over. Then, several Domain Champions were pulled to critical application projects. This turnover resulted in a loss of vision and focus. Management could have planned these changes more carefully to reduce their impact.

Executing the Vision. Initially, the core Framework Team was responsible for most of the framework development. During the later stages of Project BLUE, developers from different application groups were brought in as framework developers, often times unofficially and without being co-located with the Framework Team. This model should have been used more consistently and earlier during the project [35]. Other practitioners "advise resisting the temptation to create 'component teams' that build reusable frameworks in complete isolation from application teams. We have learned the hard way that without intimate feedback from application developers, the software artifacts produced by a component team won't solve real problems and will not be widely reused" [19].

Application developers brought to the Framework Team their domain expertise and the lessons they learned using the framework in their applications. When they returned to their application groups, they helped spread a shared vision amongst other developers, and overcome mistrust issues between application groups and the Framework Team. In a few cases, the approach backfired and decreased the stability of the framework when developers were pulled to other projects after only a short stint in framework development.

Less frequently, framework developers were moved to application groups through official internal transfers which had several visible effects. One of the first framework application was experiencing repeated delays. It finally shipped after one of the former framework architects took the helms of the application architect role. Similarly, when the author moved to an application group, he helped the other application developers better understand and use some of the areas of the framework that he had worked on earlier.

Pilot Projects. The BLUE Framework Team went from having no input at all for several months to having 20 developers literally show up at its doorsteps and conduct on-site pilot projects that lasted several weeks. The pilot developers were organized into five pilot projects, each exercising a different domain. For instance, one pilot project was interoperating the framework with a 20-year-old application that only accepted flat files as input. Another dealt with datasets large enough to strain a garbage collector.

The pilot projects started with several days of developer training where morning presentations were followed by hands-on labs. Then, each pilot team worked on its project. Every pilot team presented weekly their status and any pending issues to the other pilot teams and the Framework Team. The pilot teams used the War Room concept [32] and pair programming and were very productive. One team that did not stay onsite for the duration of the pilots had less impressive results compared to the other pilot teams. Having the pilot developers co-located with the core Framework Team and away from their usual work location helped fully dedicate their attention to the evaluation of the framework. During this period, the Framework Team implemented many change requests at a record pace to avoid delaying the pilots.

One may argue that these pilot projects were actually toy projects and that the first applications built on the framework were the real pilot projects. Indeed, "a toy project will not really challenge the architecture or the development method. Observers may not believe that the architectures will actually scale to the requirements of 'real' projects" [38, p. 82]. However, these pilot projects made more extensive use of more features of the framework than the first real framework applications ever did. In fact, the first application made a rather perverted use of the framework — what one application's project manager called a "customization of the BLUE Framework for us". In that case, the Framework Team instantiated the framework for the application and spent time developing other application-specific code.

The pilot projects were instrumental in validating the BLUE Framework while it was still under development. Using pilots is an expensive validation technique if one considers the costs of pulling 20 fulltime developers from other projects and dedicating them to the evaluation of an incomplete framework. However, not doing so can be even more expensive in the long run as it may lead to rolling out an unsuitable framework. In short, using pilots for evaluating and validating frameworks is highly recommended.

The pilot projects had the following effects [38, p. 81]. First, they exposed many weaknesses that the Framework Team needed to address. In some cases, it was apparent that the expertise to solve some of these problems was present in some of the application groups and not in the Framework Team. Second, management was able to gauge their confidence in the approach. In response, management decided to reorganize and geographically distribute Project BLUE into several sub-projects based on the core competencies of the various product centers. Third, the Framework Team now had access to the five prototypes that the pilots had developed. Unfortunately, these applications used an early version of the framework. In preparation for one of the framework training sessions, the Framework Team ported one of the pilot prototypes to the current version of the framework. That

prototype served as both a good sample framework application and a good framework unit test. Unfortunately, this was done for only one of the pilot prototypes and one version of the framework. In retrospect, the Framework Team should have ported and maintained most of the pilot prototypes as samples, across framework versions. But some of these prototypes were harder to port and maintain since they wrapped significant portions of legacy code. Finally, Project BLUE failed to gather effort data from the pilot projects for future project planning.

Documentation Matters. "Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation" [29].

Upon adopting "proof by demonstration", Project BLUE kept the documentation to a moderate level to avoid the earlier trap where "specs are stale baked [. . .] and requirements change faster than the voluminous details can be updated" [3]. However, coordination required documentation, and even more so as Project BLUE became distributed.

The Framework Team built a custom web-based tool to manage the framework use cases collaboratively, following a specific format. The Framework Team also produced detailed requirements documents for some of the better understood framework subsystems. The Product and Usability Champions formally reviewed these documents.

The Framework Team quickly realized the value of architectural documentation when different application developers started learning the framework and asking many similar questions. The framework architects devised a document template and used it to document the core framework architecture. Each framework developer working on a framework subsystem then used the template to describe the detailed design of his or her own subsystem. Hierarchy was achieved using separate "tabs" in the same document.

The description of the architecture consisted of informal boxes and lines and not a formal Architecture Description Language [2]. But it was not completely free-form. The technique adapted ideas from the CRC methodology used in object-oriented design [5]. Each Component had a list of Responsibilities (the services it provides), a list of Collaborators (the other components it either provides services to or requires services from) and a Rationale (Why is it needed? Why is it a separate component?).

Application architects then used the same template to describe the applications under development. Documenting the architectures of the framework applications this way helped the Framework and the application teams agree on how to map an application's architecture to the framework's reference architecture and to design any additional extensibility and variability points.

Documenting the framework code was paramount for coordinating work among developers. The best documented

APIs were the ones that were documented before or as they were implemented. Otherwise, they were never documented or were documented poorly just prior to a release — leaving out the important summary descriptions and overviews [37, pp. 36–38]. The Framework Team could have done better in that area, perhaps using pair programming or more frequent code reviews. Although the compiler can warn if a public class member is not documented, only a human reader can determine if the documentation is satisfactory.

In the "Mercenary Analyst" organizational pattern, Coplien and Harrison advocate "hiring a technical writer who is proficient in the necessary domains but who does not have a stake in the design itself" [14] to capture the design. This strategy was used to document an important component of the framework shortly before its author was transferred to another department. The strategy seemed only moderately successful. In that case, it might have been more effective to transfer that knowledge to the Framework Team directly.

Conway's Law. Conway's Law [12] states that any piece of software reflects the organizational structure that produced it. In a framework project, just as in any project, the lack of coordination can lead to acute manifestations of Conway's Law, and Project BLUE had many such examples.

Example 1: Visualization. Two architects designed independently two graphical visualization subsystems, one for 2D views and one for 3D views. These two designs were completely different. Each design had different strengths and weaknesses. The 3D design was powerful but complex, the 2D design simpler and more specialized. The inconsistency between these two designs was confusing for the early consumers of the framework, and in particular, the pilot projects. Furthermore, these two designs were never reconciled for several reasons. The 3D view was needed by several of the first applications targeting the framework. On the other hand, the 2D visualization languished since there was only one 2D view of interest that was needed by the first few applications. In the end, a third-party component was customized for the 2D functionality and it largely dictated the design. Finally, the other 2D views which made use of the 2D visualization had no immediate clients.

Example 2: Domain Objects. Two architects working at two different locations designed two competing implementations of a critical framework subsystem for *domain objects*, i.e., business logic. One implementation was nearly feature complete but lacked various extensibility features that the applications needed. This implementation came with a code generator to generate the programmable domain objects for the full shared application data model. The competing implementation was more extensible but supported only a small footprint of the data model. This infighting and lack of coordination slowed down significantly delivering to the application groups a full-featured and extensible implementation of the subsystem for hosting application domain objects.

Inverse Conway's Law. A framework imposes *inversion of control* on its client applications [25]. This can lead to a similar characteristic between the organization developing the framework and the organization developing applications on the framework. As the result of an *Inverse Conway's Law*, the organization must mirror the inversion of control structure of a framework, and have additional coordination between the framework groups and the application groups.

When the Corporate Risk Manager visited the Framework Team, he accurately recognized the big architectural risk of pushing the application integration strategy from the *repository* level to the *memory* level, i.e., the domain objects layer. Indeed, that critical subsystem remained problematic for much longer than anyone had anticipated.

Moving to the application group made that realization even more acute. The BLUE Framework did not initially provide a sound solution for applications to extend the framework domain objects to include application-specific features. A centralized model was adopted, one where the Framework Team managed centrally the definition of the domain objects. In a distributed model, each application managed its own extensions, and as result, applications did not integrate well at the repository or shared data model.

The BLUE Framework did not support extending the data model easily. For the sake of expedience, application developers often added code to the framework to customize the domain objects to meet specific application needs. This effectively split an application's business logic between the framework baseline and the application baseline. This unnecessary tight coupling between the framework and the applications would give rise to serious problems when more than one application used the same framework version and made incompatible extensions to the shared domain objects.

Organizational Coordination. Introducing a framework is a fundamental change to the structure of the software that binds together many applications and infrastructure that may be independent prior to the framework's introduction. This often means that an organization cannot make just a few minor adjustments and expect to achieve the productivity gains promised by product lines without dealing with the additional required coordination across the organization.

Project BLUE left organizational coordination largely unaddressed. For instance, the product champion organization was already in place for regular application development, and was left intact. In addition to their previous responsibilities, the Product Champions hammered out with the framework and application architects, the difficult factoring of functionality between what is handled by the framework and what is left to the applications. But this organization seemed to work with varying degrees of success.

The project manager for the core Framework Team kept reminding everyone that the role of a dedicated product line "project director" was needed. But the project did not create that role until much later — because it did not fit well on the organizational chart. The director organization would sit above the application groups and serve as a conflict resolution mechanism. The Chief Software Architect did not have enough clout to perform this role without sounding partial to Project BLUE, and the application architects reported only loosely to him. Even the VP of Product Development did not have the knowledge and the clout to tell an application group that, yes, they have to use the BLUE Framework, but they will have to wait for the feature they want, and bear the responsibility of the product being late or incomplete.

Fafchamps discussed how the resistance to reuse is often a symptom of weak collaboration inherent in the divisional organizational structure: "a successful reuse program begins with a careful understanding of an organizational structure, the culture that structure fosters, and the strain the reuse program may impose on that culture " [18].

In summary, the organization needed to reflect the fundamental change, the new, far-reaching interdependencies that the framework introduced, but it did not. It created a few new roles — the various Framework Champions — and tried to move forward while keeping the organization largely the same. The application groups still operated largely independently — unilaterally vetoing features and threatening to walk away from the framework. In some cases, they exerted pressure on management to be exempt from using some framework features, citing strategic technical concerns or brandishing the specter of project failure.

The organization where Project BLUE took place was not particularly unable to deal with a framework project. In a talk on the role of design in software development, Buxton argued that "when new products come out of a company, more often than not, they come out of skunkworks, they come out of illegal bad behavior that happens to turn out right" [8]. Project BLUE did many things "wrong", such as abandoned process at times, yet it managed to build a usable framework. Indeed, skunkworks allow unprecedented efforts to move ahead while insulating them from rules and practices. Business as usual might kill such projects otherwise. Keeping Project BLUE under wraps at first was perhaps a wise decision, despite the drawbacks discussed earlier.

Even if the organizational coordination issues are addressed effectively, framework projects are still hard because of the difficulty of finding the right abstractions. The next section revisits from a technical perspective some of the coordination issues discussed in this section.

4. Finding the Right Abstractions

Pree claims that "most [. . .] frameworks are still the product of a more or less chaotic development process typically carried out in the realm of research-like settings" [20]. Indeed, the development of the BLUE Framework seemed disorganized at times partly due to a lax process discipline, various process mistakes, or a lack of coordination, but mostly due to the nature of building a framework.

Framework development seems harder than regular application development because it is more difficult to find the right *abstractions* to build the right framework. The term *abstraction* is used broadly to include domain models, framework use cases, architecture, and detailed interface design, among others. The difficulty in finding the right abstractions has implications on the process itself, such as requiring iteration to refine some of these abstractions.

Clearing the Fog. Knowing what to build is difficult when building a framework for a product line. In retrospect, Project BLUE spent too much time in "paralysis by analysis". Similar results could have been achieved with less effort or time by following a strategy of "clearing the fog". Cockburn describes the approach as "do something (almost anything) that is a best initial attempt to deliver some part of the system in a short period of time. The difficulty is that you don't know what it is that you don't know. Only by making some movement can you detect what it is you don't know. Once you come to know what it is you don't know, you can pursue that information directly" [11]. Project BLUE learned to use that strategy effectively in several areas from selecting framework use cases to designing components.

Building Domain Models. Researchers have realized that "frameworks and domain models are close relatives [. . .]. The core of frameworks and of domain models is (1) to describe commonality and variability and (2) to support the management of the different parts in similar products coming from the same domain" [20, p. 213]. The Framework Team found domain modeling difficult. This resulted in longer than planned requirements activities, missed schedules and inadequate functionality, especially during the early stages of framework development. Domain modeling seemed particularly challenging for Project BLUE for the following reasons:

- **Lack of business domain modeling:** the Product Champion produced a detailed business use case model for only one of the sub-domains. In retrospect, the Product Champions should have been able to produce detailed domain models for each of the well understood sub-domains;
- **Lack of domain experience and domain modeling experience:** most of the core Framework Team developers did not have application or domain experience in more than one domain and had limited expertise in domain modeling techniques;
- **Lack of stakeholder validation:** Project BLUE initially kept a low profile and did not seek stakeholder validation during its Inception and Elaboration phases;
- **Broad domain and scope creep:** the BLUE Framework had a scope that was too broad and kept increasing as the project added more stakeholders. Perhaps, it was unrealistic for a single framework to address such a wide variety of domains.

Identifying Framework Use Cases. Having to derive framework use cases from a number of application use cases is another reason why framework development is harder than regular application development. Indeed, this task seemed particularly challenging to the framework developers. In some cases, the Product Champions identified some of these framework use cases. In other cases, the more senior framework developers who had worked on several applications harvested use cases and designs from prior applications. Finally, "clearing the fog" (as described by Cockburn) and serendipity played an important role.

When identifying framework uses cases, Project BLUE committed the serious process mistake of letting the scope of the framework and of each iteration get bigger, and this introduced delays at various points. There were several reasons for the scope creep:

- **"Pork barrel" features:** in some instances, certain application groups threatened to boycott the framework if certain capabilities were not present. In order to ensure their buy-in, the framework architects succumbed to those pressures by increasing the scope.

  When multiple visionaries are involved, there's the temptation for each Product Champion to propose his or own set of favorite requirements. The architects, together with management, often performed the balancing act, and made sure that specific business drivers dictated each framework feature request. But a more appropriate slogan would have been to "deliver less, more often" [36].

- **Gold-plated requirements:** some requirements were initially considered critical, but were later considered gold-plating, i.e., just "nice to have".

  Example: Scripting. Adding scripting capabilities to the framework was one such example. Initially, a comprehensive scripting third-party component was integrated. However, the solution was heavyweight and had restrictive licensing requirements. This implementation was later replaced with the simplest thing that could possibly work. This meant using a primitive script editor without syntax highlighting, auto-completion, etc., and not having a script debugger. The reasoning was that only a few power end-users were going to write scripts anyway. In most cases, commonly used scripts would be deployed with the applications and would be only customized by most end-users.

- **Extreme requirements:** *extreme requirements* impose undue demands on the entire system, and must remain under challenge throughout system design, implementation and operation [30].

  Example: Collaboration Services. One extreme requirement was to have the BLUE Framework provide primitives for writing collaborative applications. This functionality required many far-reaching changes to the core framework to support in-memory transactions, locking, change notifications, etc. This requirement was initially considered a critical market differentiator but then was deemed a non-requirement for the first few applications

targeting the framework. After significant effort was spent on this functionality, it was left partially implemented and abandoned.

- **Strategic shifts:** The decision to build a product line is not likely to prevent acquisitions — and the software integration challenges they bring, while the product line is being institutionalized. Indeed, midway through Project BLUE, the Company acquired a platform that significantly affected the technical architecture and future plans of the BLUE Framework.

Architecture-First Approach. One of the hardest technical challenges faced in framework development is creating an architecture that, for many years to come, will serve as the basis for several products, as well as inter-operate with legacy systems. Designing the right framework architecture is another instance of coming up with the right abstraction. The Rational Unified Process advocates the *architecture-first approach* to deal with risks such as late breakage and excessive rework and necessary technology insertion [33].

Overall, Project BLUE did this reasonably well by assigning strong architects to the Framework Team. Fowler identifies two sub-species of architects [21], and these two were explicitly represented on the Framework Team: 1) a *macro-architect*, who makes all the high-level decisions; and 2) a *micro-architect*, who has a greater attention to detail, while still being aware of all the high-level decisions.

The Macro-architect. Project BLUE had a dedicated software architect assigned to it. The Macro-architect's official job title was "project manager", however, he left most of the daily project management to first-line managers. The Macro-architect was a strong architect as demonstrated by his promotion to Chief Software Architect of the software organization a few years later. Indeed, Project BLUE got off to a better start than most transition projects. Such projects are often staffed with new hires and external consultants because everybody is busy with urgent development projects [38, p. 80]. Assigning one of the most respected project leads to head Project BLUE was an indication of management's commitment to the product line.

In addition to making high-level decisions in close consultation with management, the Macro-architect was writing code to help mitigate some of the architectural risks. In particular, he evaluated third-party components in his areas of expertise. Coplien and Harrison call this the "Architect implements" pattern and add that "too many software architects limit their thinking and direction to abstractions, and abstraction is a disciplined form of ignorance" [13].

The Micro-architect. The Macro-architect left most of the detail-oriented tasks to the Lead Architect on the Team, who represented the Micro-architect role identified by Fowler. The Micro-architect also served the role of the Chief Programmer [1]. He often sat down with the core framework developers and made sure that the individual parts integrated well with the overall design. He wrote large por-

tions of the core code himself. The Micro-architect traveled often to proselytize the use of the framework to the various application architects and developers. Needless to stay, the Micro-architect had high demands placed on his time, and did not have much time left to write as much code as his Chief Programmer role would require. As a result, he may have been on the critical path when the Framework team required a critical feature to be implemented or a serious bug to be fixed.

The Local Architects. Coplien et al. recommends having "an architect at each location. Architects can be the focus of local allegiance, which is one of the most powerful cultural forces in geographically distributed development" [14]. However, this approach was not without problems; it led to infighting and double work in the case of the subsystems for visualization and domain objects.

It is also critical during the transition to a product line to assign strong architects to application groups building applications on the new platform. For many of the early framework applications, Project BLUE seemed less fortunate in that area. Indeed, every time this wasn't the case, the application project faced serious troubles. As discussed earlier, one of the first framework applications meandered aimlessly for several months and fell significantly behind schedule. The application did not get completed until one of the former framework architects was temporarily assigned to the project as the application architect. Ultimately, he was able to mentor another team member to take over that role.

The author acutely experienced the need for good architectural guidance after he moved to a group building an application on the BLUE Framework. Several of the application developers privately complained that due to the lack of a clear architecture, the application consisted of a hodgepodge of features. For instance, each screen in the application looked like it had been developed by a separate developer, which was indeed the case. A *macro-architect* for that application could have prevented this. In addition, the application code was poorly written and had many instances of duplicated code. Many interface method signatures were not type-safe, which prevented developers from catching most of the errors at compile time when the underlying data model changed. A *micro-architect* role could have prevented or addressed such problems.

Reviews. The goal of evaluating the architecture is to reduce the levels of scrap and rework and to avoid having a design that reflects the organizational chart.

In one of the early framework releases, the various subsystems followed different architectural styles since each one had been worked on independently. The framework architects then held several internal architectural reviews to outline several important architectural and design principles that all developers had to abide by.

For example, there are two major modes of framework use: 1) *white-box reuse*, where the main technique is inher-

itance; and 2) *black-box reuse*, where the main technique is composition. The framework architects strongly encouraged the use of composition and discouraged the use of inheritance to avoid the *fragile base class problem* [28]. This way, renderers for 2D and 3D visualization as well as views and dialogs for the shared application container used the same plug-in architectural style.

These internal reviews discovered several important defects, but not all. More thorough architectural and design reviews, as in the Architecture Tradeoff Analysis Method (ATAM) [9], earlier in the process, could have spotted additional design defects sooner. For instance, a simplistic design that relied on passing a single "baton" was initially adopted to support the collaborative primitives. Later in the project, this model was considered inadequate and abandoned in favor of in-memory transactions. The original weak design may have resulted from the "clearing the fog" strategy discussed earlier. As Cockburn put it well, "if you only 'clear the fog' and 'clear the fog' [. . .], you will not make real progress. You will have lots of little experiments and no deliverable results" [11].

The application architects also discovered important framework defects. For example, the Framework Team initially proposed a notion of "automatic save", which committed any changes to the data automatically. But some applications wanted the document metaphor commonly used by productivity applications where the user can open an existing document, make changes to it, explicitly save any changes or discard them.

Architecture Bootcamps. The Framework Team established a series of architecture bootcamps consisting of architecture training sessions and workshops. In Inception, an application group would receive training on the framework. In early Elaboration, an application group would receive one-on-one guidance to make sure that all the required capabilities were already implemented or under construction in the framework. Finally, in late Elaboration, the application architecture was assessed to evaluate its use of the framework.

The purpose of the architecture evaluation meetings was twofold. First, the framework architects helped application architects understand how to make better use of the framework architecture. Second, the application architects helped the framework architects determine if the framework was missing any critical features, or if there was any need for architectural refactoring.

The architects often traveled for these face-to-face meetings, which compensated for the geographic distribution and improved the coordination. But, the framework was still under development and required their attention. As a result, the framework architects became thinly spread. Shipping the framework architects with the framework to ensure its correct usage was certainly not a scalable approach. Perhaps, this was an indication that the framework was too complex or that the documentation was lacking.

Organizational Impact. Change is slow and difficult in most organizations. Project BLUE faced chronic organizational inertia with several important vocal detractors. Maintaining "business as usual" could have potentially killed the project. Generally, application groups do not want to lose their autonomy. They enjoy being able to make things happen without relying on other groups, and believe they are the best people to deliver their product [27].

"A product family approach involves more than architectural issues alone; it also affects a company's business, processes, and organization" [40]. Indeed, Project BLUE had to take into account many extra-technical considerations when making the various technical decisions. These considerations involved implications for licensing, intellectual property, product branding and deployment. Some of these considerations were initially neglected, and led to much frustration and rework later in the development cycle.

Harvesting Design. "Interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or re-create than code" [16]. Nowhere is this more palpable than in a framework project which is mostly about designing interfaces.

To come up with the right abstractions, "either the *problem domain* is analyzed to create a new design, or the *solution domain* is analyzed to understand how the problem has already been solved. Solution domain analysis is advantageous for at least two reasons: solutions are likely to have addressed more concerns because they have been tested in a real-world environment and they provide a source for reusable code. On the other hand, such solutions may be too implementation-specific so as to inhibit code or design reuse. The objective of harvesting is to extract as much design and code as possible from existing solutions so that it can be refined for reuse" [6].

Architecture bootcamps harvested design from the applications and incorporated their commonalities into the BLUE Framework. However, Project BLUE could have harvested more design. For instance, Project BLUE took three iterations to produce an application hosting and task management subsystem. In fact, a similar design had been previously documented [6]. Perhaps harvesting that design would have allowed the BLUE Framework to converge on the appropriate design faster. Other fertile grounds for harvesting design include some of the well-designed open-source frameworks.

Keeping the Core Small. The way the BLUE Framework evolved seems to confirm that it is often sufficient to write most parts of a framework in terms of small number of interfaces (i.e., contracts) and patterns and use these patterns consistently [24]. For instance, the first version of the shared application container included many hard-coded menu items in the top-level window and only a few extensibility points to add menu items in a view-specific toolbar using the Command design pattern [22, p. 233]. The later versions of the application container did not contain a single

hard-coded menu item — even the one to quit the application was optional, and each menu item was added explicitly. That version used the Command design pattern consistently throughout. A command could be added as a view-specific command in a view-specific toolbar, or as a top-level command as a menu item or a toolbar item. Using known design patterns also makes documenting the framework easier [24].

Maintaining Compatibility. It is difficult to eliminate or change a framework interface once applications deployed on the framework start using it. For instance, the framework architects wanted to change the design of the domain objects layer. But they were reluctant to generate the domain objects layer for the entire data model to avoid having to support it. Different applications at various stages of completion were already using different pieces of the shared data model with their own extensions. As a result, each of the early applications was frozen on one version of the framework and one instantiation of the data model.

The Framework Team was accommodating for the first few applications built on the framework while they were under development. After their critical initial releases, these applications were still reluctant to upgrade to a newer version of the framework, since it would have required significant changes to the applications. This situation will quickly become untenable, and the Framework Team will have to stop supporting these early framework versions. This was another reminder of the "Preservation of Pain Principle" coined by the framework Micro-architect: either constantly synchronize the applications to the same version of the framework or have one version of the framework with many backwards compatibility features.

Managing Optional Features. "Frameworks fail miserably in the very common case of optional features" [4]. There was a strong temptation to add too many features to the core framework — over-featuring was perceived as a way to avoid duplicating code for framework instances. For this reason, the framework architects strongly preferred interfaces over abstract classes since the implementation language did not support multiple class inheritance or implementation inheritance. Implementing the framework in terms of interfaces produced swappable implementations for applications with significantly different needs. In the end, the framework consisted of a small set of interface definitions and default implementations of those interfaces.

Managing Dependencies. Framework developers often introduced excessive dependencies during the development cycle. Although every effort was made to keep the test code separate from the framework code, in one case, one core framework component depended on some utilities from unit testing code. In another more extreme case, a developer broke the overnight build by adding a circular dependency just by referencing an icon from another project. A component may become less reusable if it has excessive dependencies on other components, but unfortunately there was no

mechanism to keep the dependencies in check. As a stopgap solution, the Framework Team relied on dedicated "cleanup" phases to remove the extra dependencies before important release milestones.

Designing for Change. The technology framework underlying the BLUE Framework came with user interface controls that offered only minimal capabilities. However, many framework applications were requesting additional bells and whistles.

To focus the limited resources on the important business-specific functionality, the Framework Team relied on third-party components to provide these additional features. The framework was designed for change by wrapping most of the third-party components exposed to the applications. This enabled swapping the underlying libraries providing widgets (such as toolbars and tree controls) several times without breaking any of the framework applications — the latter only reflected the enhanced look-and-feel.

Ensuring Framework Compliance. Enforcing the compliance of an application to a framework is difficult. Framework compliance goes beyond ensuring that a given application is using the framework API correctly. Successfully creating a framework application requires clearly understanding the framework architecture and following the specification for extensibility, to become a "good citizen" in the framework ecosystem. Project BLUE did not have a good solution for enforcing that policy.

The BLUE Framework architects and the Product Champions produced a "compliance document". The document insisted on several fundamental principles that each application must adopt, but allowed some selectivity in the more optional aspects. In comparison, the original vision document — that the Product Line Champion authored — mandated more rigid uniformity.

Unfortunately, many developers preferred a pick-and-choose strategy and treated the framework as a toolkit — a mode of use that does not ensure a consistent look-and-feel and encourages gratuitous differences. Institutionalizing a product line often requires overcoming perceptions of uniqueness. Others have reported how entrenched that perception is. "The core asset group worked patiently with both sides to capture the details of the two applications, and at the end of the exercise, it turned out that the two features were not only similar but in fact functionally identical, modulo a numeric constant" [2, p. 359]. In Project BLUE, there were many such examples. One application group was threatening to not use the shared application container if it did not support docking toolbars vertically as well as horizontally. The Framework Team switched to a different third-party library with that feature to eliminate such a frivolous excuse for not using the framework.

Iterating Framework Design. "Good frameworks are usually the result of many design iterations and a lot of hard work" [41] and there are several reasons for this [20, p. 21].

Project BLUE did one thing well, and that was refactoring mercilessly instead of shoehorning late fixes onto the framework. Merciless refactoring was used to eliminate error-prone or brittle parts of the design, as well as change confusing interfaces. The BLUE Framework project manager had a healthy attitude towards iteration and refactoring and thought it was never too late to do the right thing.

A risk averse project manager would have prevented developers from performing far-reaching refactorings such as renaming classes and methods. In contrast, developers in the Framework Team were encouraged to refactor even as major deadlines approached, on one condition: "you break it, you fix it". This was not as trivial as it sounds since the tools in use at the time did not support many refactorings (e.g., capture avoiding substitutions instead of simple string rename). In many cases, framework developers had to directly notify application developers of known breaking changes, offer to fix their broken code, and even convert their configuration files from the older to the updated format.

In comparison, when the author joined the application group, the application project manager seemed more averse to refactoring, although there were several modules that could have greatly benefited from refactoring. Application groups typically work with accelerated schedules, whereas framework projects are not tied to a particular schedule, so this may explain this difference. Furthermore, the problem of brittle code is more severe for a reusable, long-lived, widely-used framework than for an application.

Example. In early versions of the BLUE Framework, too much functionality was encoded in external configuration files. This lead to a brittle design when the wrong settings produced silent failures which were hard to debug. In subsequent iterations of the system, the Framework Team moved away from that model and used two strategies instead:

- Use source code annotations for the meta-data information that could be checked statically by a compiler, e.g., use `typeof(ClassName)` instead of referring to the name of the class with the string `"ClassName"`; and
- Place in external XML files only minimal information such as the fully qualified type name of the plug-in class. A plug-in did the rest of its initialization when its main class was instantiated.

Unfortunately, many widely used open-source frameworks have not heeded this lesson.

5. In Retrospect

Framework Introduction. Building a product line is a medium-to-long term goal, so it may not be possible to show quick results. It has been reported that it takes three to four times longer to build a product line as it does to build an individual application [20]. However, it is important to release early and often. This is the only way to make sure that a framework can support the applications. It is a big risk for applications to start using a framework while it is changing.

But, the only way to find out what is wrong with a framework is to use it. For the first applications contemplating the use of the BLUE Framework, the architects had to do a lot of hand-holding to promote the adoption of the framework. They also had to resign themselves to be blamed for many problems. In many cases, application groups experiencing the slightest turmoil blamed some of their problems on the framework. In addition, the first few applications did not make heavy use of the BLUE Framework. The first application used only a few features of the framework. The second one had only a slightly larger footprint. Each application made limited use of the framework in its own different way and was developed on a different version of the framework. In addition, many of the framework APIs and internals were significantly different across versions.

Revolution vs. Evolution. Although the author did not remain at the Company long enough to see how the product line and the framework ultimately turned out, the first few years were a mixed success. For instance, it was hard to believe that anyone expected the project to take this long and this many developers.

There are various strategies for introducing product lines: "in the ideal approach, large investments are made initially (*revolution*) and recovered later. This is, however, a risky undertaking since this approach may fail because it costs too much and/or takes too long. A more realistic scenario assumes *partial, stepwise introduction* without too much impact. This will lower the risk and result in benefits at an earlier stage. Feedback is obtained at an earlier stage this way and acceptance can be built up gradually. However, the benefits may not increase as fast as in the ideal situation, because the approach will have to be improved gradually over time" [40, p. 83].

In this case, stepwise introduction may not have succeeded either, since the first applications built on the framework had limited integration with each other and achieved different look-and-feel results for end-users.

Build vs. Buy. Ralph Johnson once said: "First Rule of Framework Design: Don't. Buy one, instead" [23]. Looking back at the early days of this project several years later seems to confirm the intuition behind that quote. Developing frameworks is hard and takes a long time. Many of the surrounding technologies may independently evolve and mature during the time it takes to a build a framework. For example, the underlying technology framework significantly evolved while the BLUE Framework was under development, and came to include many features that Project BLUE built in-house. The market may have never produced the complete BLUE Framework. But the maturing over time of the infrastructural technologies could have possibly reduced the amount of functionality that Project BLUE needed to implement from scratch.

Of course, waiting has an opportunity cost as well. The Company followed both approaches — building and buying.

First, it started and continued development on the BLUE Framework. Second, it acquired a strategic platform which was rapidly gaining market share. The acquired platform enabled the continued development and enhancement of award-winning applications. But the platform was not truly a framework; in particular, it did not serve the needs of all the application domains that the BLUE Framework intended to support.

A recent trend in the industry suggests another strategy to get the best of both building and buying: use available open-source frameworks whenever possible, instead of building from scratch custom frameworks. For instance, the Eclipse Project now fully supports the Eclipse Rich Client Platform (RCP), after several companies used the Eclipse Framework to build general purpose applications and not just development tools [17]. The Eclipse framework enjoys a large community of developers and offers an open-source friendly license. As a result, the Eclipse RCP framework built on top of the Java technology platform seems like an ideal building block for an enterprise framework.

Frameworks = Coordination. There are many hard organizational and cultural issues that must be addressed in order to yield optimal results. So frameworks require dealing with organizational coordination in addition to the difficult technical problems such as interoperating with legacy systems and third-party components,

Project BLUE was promising application developers a "step change in productivity" through reuse and extensibility. However, a new framework with a new technology did not change the underlying stovepipe mentality nor did it change the organization. For instance, even after Project BLUE was launched and the first few applications on the BLUE Framework were released, there was still a separate data management group who had monopolistic control over the centralized data model.

The step change in productivity largely failed to materialize because the organization stayed the same. And as a result, the structure of the software remained the same. For instance, in order to implement certain applications requirements, several things had to take place. First, the Data Modeling Team extended or modified the data model. Then, the Framework Team re-generated the domain object layer. Finally, the application developers implemented the application-specific business logic.

Many of the benefits that the Company aimed to achieve with the BLUE product line such as better integration and consistent look-and-feel failed to materialize early on, because the organization did not change enough to support these ambitious goals. The first few BLUE Framework applications looked as different and had as little integration as if they had been developed using different platforms.

Afterword. Although Project BLUE faced internal resistance within the organization, many of the external consumers saw the value of the product line and embraced the idea with enthusiasm. This helped offset the reluctance that some of the internal application groups had in adopting the new technology and the constraints that came with it.

6. Conclusion

Building a framework is a once- or twice-in-a-career opportunity for most technical people, at all levels of leadership. As a result, framework development will continue to be hard until a significant body of knowledge for building product lines and frameworks is documented, and organizations learn how to deal with these kinds of projects. By focusing on the most significant issues and pitfalls that Project BLUE encountered, discussing how it addressed some of them successfully and what it could have done to address some of the others, this experience report is intended as one such contribution to the field.

To summarize the foregoing lessons:
1. Use proof by demonstration, with judicious fireworks;
2. Clear the fog, but don't only clear the fog;
3. Conduct pilot projects — they are worth every penny;
4. Release early and often;
5. Keep the vision sharp and well-defined;
6. Manage the architecture as the most crucial asset and communicate it well;
7. Assign good macro-architects and micro-architects to the crucial product line projects;
8. Harvest as much design as possible — including designs from open-source frameworks;
9. Iterate and refactor mercilessly, until you get it right;
10. Heed Conway's Law and its Inverse: the software and the organization mirror each other.

Acknowledgements

References

[1] F. T. Baker. Chief Programmer Team Management of Production Programming. *IBM Systems Journal*, 11(1):56–73, 1972.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[3] P. G. Bassett. *Framing Software Reuse: Lessons from the Real World*. Prentice-Hall, 1997.

[4] D. S. Batory, R. Cardone, and Y. Smaragdakis. Object-Oriented Frameworks and Product Lines. In *Software Product Lines (SPLC)*, pages 227–248, 2000.

[5] K. Beck and W. Cunningham. A Laboratory for Teaching Object-Oriented Thinking. In *Object-Oriented Programming Systems, Languages and Applications*, pages 1–6, 1989.

[6] J. Boone. Harvesting Design for an Application Framework. In *Conference of the Centre for Advanced Studies on Collaborative research*, 1996.

[7] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*. Morgan-Kaufman Publishers, 1995.

[8] B. Buxton. The Role of Design in Software Product Development. `http://www.billbuxton.com/`, 2004.

[9] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture: View and Beyond*. Addison-Wesley, 2003.

[10] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[11] A. Cockburn. Project Risk Reduction Patterns. Technical Report HaT.TR.97.02, 1997.

[12] M. E. Conway. How Do Committees Invent? *Datamation*, 1968.

[13] J. O. Coplien and N. B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice-Hall, 2004.

[14] J. O. Coplien, N. B. Harrison, and G. Bjornvig. Organizational Patterns: Building on the Agile Pattern Foundations. *Cutter Consortium Agile Project Management Advisory Service*, 6(6), 2005.

[15] T. A. Coram. Demo Prep: a Pattern Language for the Preparation of Software Demonstrations. In J. Vlissides, J. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 407–416. Addison-Wesley, 1996.

[16] L. P. Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. In *Software Reusability: Volume II, Applications And Experience*, pages 57–71. Addison-Wesley, 1989.

[17] Eclipse Rich Client Platform (RCP). `http://www.eclipse.org/rcp/`.

[18] D. Fafchamps. Organizational Factors and Reuse. *IEEE Software*, 11(5):31–41, 1994.

[19] M. E. Fayad and D. C. Schmidt. Lessons Learned Building Reusable OO Frameworks for Distributed Software. *Communications of the ACM*, 40(10):85–87, 1997.

[20] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 1999.

[21] M. Fowler. Who Needs an Architect? *IEEE Software*, 20(5):11–13, 2003.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[23] R. Johnson. How to Develop Frameworks. OOPSLA Tutorial, 1997.

[24] R. E. Johnson. Documenting Frameworks with Patterns. In *Object-oriented Programming Systems, Languages, and Applications*, pages 63–76, 1992.

[25] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[26] P. Kruchten. *The Rational Unified Process: an Introduction*. Addison-Wesley, 1999.

[27] C. Krueger, J. Dager, U. Olsson, A. Heie, M. Verlage, and B. Hetrick. Avoiding, Surviving, and Prevailing Over Pitfalls in Product Line Engineering. In Software Product Line Conference (Panel), 2004. `http://www.sei.cmu.edu/SPLC2004/panel_slides`.

[28] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *European Conference on Object-Oriented Programming*, pages 355–382, 1998.

[29] D. L. Parnas. Software Aging. In *International Conference on Software Engineering*, pages 279–287, 1994.

[30] E. Rechtin. *Systems Architecting: Creating and Building Complex Systems*. Prentice-Hall, 1991.

[31] D. Roberts and R. E. Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison Wesley, 1996.

[32] N. Ross-Flanigan. Working Together In "War Rooms" Doubles Teams' Productivity. `http://www.umich.edu/~urecord/0001/Dec11_00/2.htm`, 2000. University of Michigan, News and Information Services.

[33] W. Royce. *Software Project Management: a Unified Framework*. Addison-Wesley, 1998.

[34] Software Engineering Institute. A Framework for Software Product Line Practice. `http://www.sei.cmu.edu/productlines/framework.html`, 2006.

[35] P. Toft, D. Coleman, and J. Ohta. A Cooperative Model for Cross-Divisional Product Development for a Software Product Line. In *Software Product Lines (SPLC)*, pages 111–132, 2000.

[36] J. van Zyl and A. J. Walker. Strategic Product Development: a Strategic Approach to Taking Software Products to Market Successfully. In *Software Product Lines (SPLC)*, pages 85–110, 2000.

[37] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, and P. Thompson. *The Elements of Java Style*. Cambridge University Press, 2000.

[38] T. Wappler. Remember the Basics: Key Success Factors for Launching and Institutionalizing a Software Product Line. In *Software Product Lines (SPLC)*, pages 73–84, 2000.

[39] K. E. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley, 2002.

[40] J. G. Wijnstra. Critical Factors for a Successful Platform-Based Product Family Approach. In *Software Product Lines (SPLC)*, pages 68–89, 2002.

[41] R. J. Wirfs-Brock and R. E. Johnson. Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9):104–124, 1990.