CHAPTER 15

Patterns for Successful Framework Development

Andreas Rüping

Introduction

Object-oriented frameworks play an important role in many IT projects these days. Frameworks allow us to reuse not only code, but abstractions and designs as well. Projects typically build a framework with the intention of large-scale reuse.

It's a difficult job, though. There are things that make framework development particularly hard. Reuse may be a promising goal, but finding abstractions is difficult. Frameworks can easily become too generic and, as a consequence, too complex and difficult to understand. Framework development can use up a lot of time and resources before it pays off.

Moreover, the context for framework development can be difficult. Ideally, a framework evolves from long-term experience acquired while building several similar applications [Brugali+1997] [Johnson+1998]. It's fairly common, however, for a software development project to decide to build a framework after identifying a potential for reuse across several applications that are going to be developed, although this means that the framework and the applications will have to be developed more or less simultaneously.

Is this even possible? Is there a chance that a framework developed in such a difficult context can live up to its promise—the reuse of code and design?

This chapter presents a collection of patterns that address these questions. It is targeted at software architects and developers who consider building a framework to meet requirements for reuse. The patterns assume an overall context in which framework and application development take place, at least to some extent, simultaneously. Many of the solutions that the patterns recommend actually apply to framework development in general, but they are particularly useful in this specific context.

The patterns address a framework's architecture, its development process, and questions of team collaboration. I have observed the patterns throughout many projects. The collection isn't necessarily complete, but it does represent a useful set of core strategies.

Project Background

I'd like to explain the patterns by using my experience with two projects that both decided to build a framework in order to facilitate application development. My experience in these projects were both positive and negative, and I'll use the projects as running examples throughout this chapter. Let's therefore first take a look at the two projects.

The Data Access Layer Framework

An insurance company faced the problem of having a large number of legacy systems that didn't work together well. The company felt it was time for a change and decided to build several new applications, including new policy systems for health insurance, life insurance, and property insurance, as well as new customer, payment, commission, and workflow systems. It was a huge endeavor involving several teams, ranging from 8 to 40 people each, working on the various applications more or less simultaneously.

Early on, the project established a team that worked on what was called the "horizontal tasks": the specification, design, and coding of modules that many—and perhaps all—other teams could use. The motivation was to save time and costs, and to ensure a consistent architecture across the new applications.

It soon became clear that providing database access would be among these horizontal tasks. All applications required database access, and in order to hide the objects' physical representations from the applications, all applications were supposed to introduce a data access layer into their architecture.

The team decided to build a framework that would provide the data access layer. The framework had to abstract over the individual applications' data models and therefore allowed application developers to specify a mapping from domain-specific objects onto database tables. The framework also provided a mechanism for object versioning that was quite powerful. Application developers could tailor the framework to the individual needs of their projects by configuring the extent of object versioning that they needed.

The framework offered the advantage that the application programmers didn't have to bother with the details of database access, including the intricate versioning mechanisms.

The Web Portal Framework

The goal of this project was to develop a Web portal for the financial industry. The portal included both Web content and applications, and was used by a bank to sell insurance products. The Web content covered general information about the available products, while the applications provided access to different insurance systems running on various back-end servers that stored and processed contracts and customer information.

Several of these applications had existed for a while when the plans for the portal were made, but a few still had to be developed. The project's main task was to integrate old and new applications into the portal. The team decided to develop a J2EE-based framework that would provide a reusable infrastructure for all applications that had to be integrated into the portal.

The framework extended over the Web server and application server layers. Its task was to take user requests, forward them to the back-end applications, transform the results into HTML and integrate them into a Web presentation. The framework also managed the use cases for the entire portal and performed the necessary session handling.

Application developers essentially had to define the use cases their application required, and to provide a mapping from client requests onto application calls. They could then rely on the mechanisms the framework provided. The definition of the use cases required a bit of programming, but the bulk of integration work was done by the framework and hidden from the application developers.

Roadmap

Figure 15–1 gives an overview of the patterns presented in this chapter and briefly sketches the relationships between them.

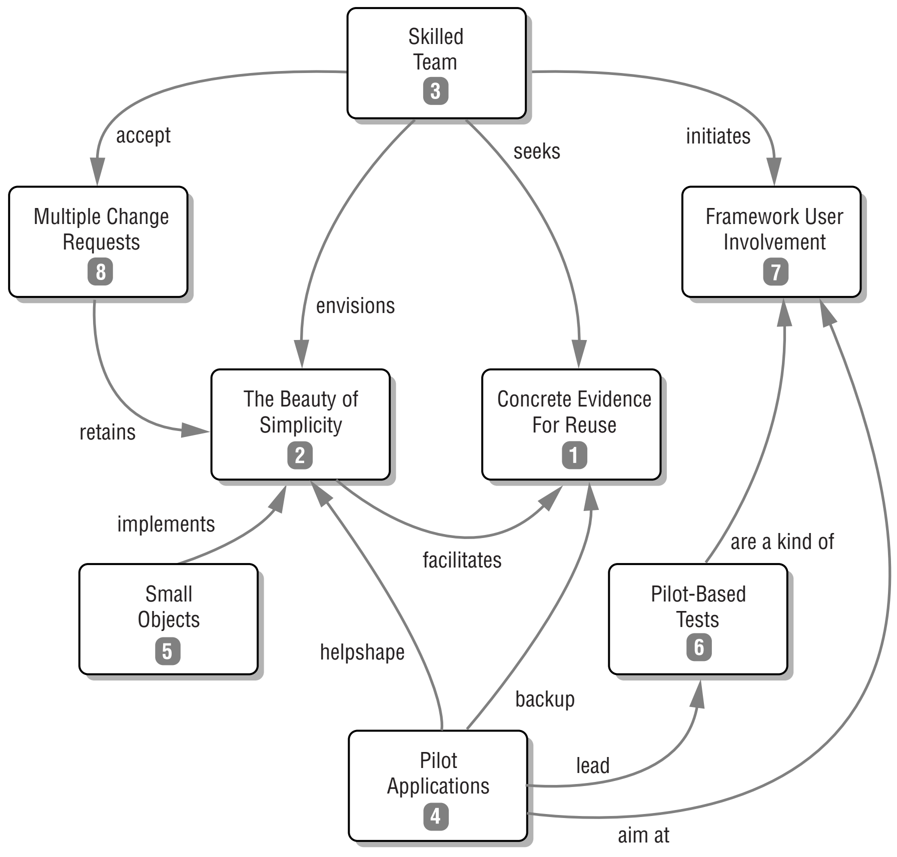

Figure 15–1 *Overview*

1. Concrete Evidence for Reuse

Problem

How can you make sure that in your project building a framework is justified?

Context

You're working on a project that will see the development of several applications. You have identified some functionality that all applications will probably require. The question of whether it would be useful to have a framework that could provide this functionality arises.

Forces

It is of course tempting to design a framework—the framework could provide a set of common abstractions and could serve as a blueprint for functionality that could then be reused many times. Building a framework can be attractive whenever there is some functionality that several applications will require, though in slightly different ways.

But reuse isn't the only aspect that lets a framework appear attractive. If you decide to build a framework, many design decisions will be made only once, leading to a consistent architecture across all applications.

Moreover, not all application developers will have to be concerned with all aspects of the software architecture if some of these aspects can be assigned to the framework. In this case, the framework designers can provide an implementation that all applications can inherit. This is particularly useful when it comes to intricate techniques that not all developers have the skill to use.

On the other hand, there is no way to deny that building a framework takes significantly more time than building a normal application. A rough estimate is that developing a framework takes about three times longer than developing an application, though the exact figure depends upon the framework's size and its degree of abstraction.

In a similar vein, the literature on reuse has a "rule of three," which says that an effort to make software reusable is worth it only when the software is reused at least three times [Jacobsen+1997] [Tracz1995]. A framework needs to be used for three different applications before the break-even point is reached and the investment pays off.

All this suggests that building a framework is justified only if the project can name at least three applications that are going to use it.

A higher number of applications sounds even more promising, but then, there is also a drawback if too many applications are supposed to use the framework: the number of stakeholders can increase to a point where reaching an agreement on the framework's scope and functionality becomes extremely difficult. The larger the number of potential framework users, the more you need to ensure that using the framework isn't just an abstract idea, but a concrete possibility.

The dilemma, however, is that you must make the decision for (or against) a framework at the beginning of the project. The application development teams must know as early as possible whether or not there will be a framework they can use. At the beginning of the project, however, the architecture of the individual applications might still be unclear. Perhaps there is even uncertainty about what applications are going to be developed. In other words, you may have to make a decision for or against a framework without knowing much about the applications that might possibly use it.

Solution

Build a framework only if there is concrete evidence that several applications are going to use it.

The emphasis here is on the word *concrete*. If all you know is that there are several applications that *might* use the framework, all you can conclude is that framework development *might* pay off. Or perhaps it won't.

Therefore, before you decide to build a framework, you must make sure that the preconditions are met:

- Three expected uses is a must. If there are four or five applications that you can expect to use the framework, that's even better, since as a project goes on, things can change, applications may be cancelled, and you may lose a potential user of your framework more quickly than you might think.
- Checking for concrete evidence includes seeking an agreement with all stakeholders of the application programs.
- The teams who build the applications must make a commitment to using the framework. This is not so much a protection against the not-invented-here syndrome, but a cross-check that using the framework is indeed appropriate. Only the application developers can evaluate how much they could benefit from the framework-to-be.

If you choose to build the framework, calculate about three times the budget you would need to build a single application. If you have concrete evidence for reuse, you can explain why the expected benefits will outweigh the costs.

Examples

The Data Access Layer Framework At first it looked as if six applications were going to use the database access framework. In the end, one of them didn't use it for "political" reasons—a consequence of teams from many companies with conflicting goals working on one large project. With five applications remaining (and more expected), building a framework was still justified.

It took the team about 30 person months to complete the database access layer framework. The team estimated that it would have taken about 12 person months to develop a database access layer for one specific application that is equally powerful with respect to business objects and versioning. (However, exact figures would depend on the size of the application's data model.) Given the fact that building an application using the framework also takes some time, three instances of reuse seemed to be the break-even point in this project.

The Web Portal Framework When the project started, the decision to build a framework was motivated by the perspective that a number of applications were going to be integrated into the Web portal. After all, the portal was supposed to offer a rich functionality to its users. Work on the framework began quickly, and soon the life insurance system and the customer system were integrated into the portal.

After a while, however, it was recognized that not as many applications were going to be integrated into the portal as had originally been planned, and that for some of these applications a less powerful solution was sufficient. The potential for reuse turned out much smaller than the project had assumed. As of now, the framework has only been used twice, and so far, hasn't paid off. The budget calculation says the framework might pay off in the future, but only if at least one more application is going to use it which, at this time, is uncertain.

Discussion

In their patterns for evolving frameworks, Don Roberts and Ralph Johnson suggest that THREE EXAMPLES [Johnson+1998] be developed before building a framework. This recommendation intends to prevent you from not finding the right abstractions or generally heading in the wrong direction with your framework—an easy consequence of a lack of experience.

Yet in many practical scenarios, there are no three examples on which you could possibly rely since the applications will be developed more or less simultaneously to the framework. Is there anything you can do instead to make sure that the framework benefits from experience with application development?

Yes, there is. Because you have concrete evidence for reuse, you must have a good idea of applications that can benefit from the framework, and how. If you choose one or two of those as PILOT APPLICATIONS (4), you can incorporate feedback from application development into the framework development.

Next, a SKILLED TEAM (3) is certainly helpful. Ideally, team members have developed a framework in such a context before. The more the team members are aware of typical pitfalls in framework development, the better.

Finally, a larger framework will have a higher risk of wrong abstractions and hopes for reuse that won’t materialize. Aim at THE BEAUTY OF SIMPLICITY (2).

2. The Beauty of Simplicity

Problem

How can you prevent your framework from becoming unmanageable?

Context

You have come to the conclusion that building a framework in your project is justified. Several applications are going to be developed and it's clear that they will benefit from the framework. You are now in discussions with the potential framework users about what functionality the framework should cover.

Forces

Obviously, the application development teams will come up with requirements for the framework. After all, they are supposed to use the framework, so the framework better meet the requirements they have. However, application developers are sometimes inclined to place more and more requirements on the framework; greater functionality offered by the framework means more time and effort the application development teams can save.

Moreover, framework developers sometimes flirt with wanting to build the *perfect* framework. Just one more abstraction here, and another generic parameter there, and the framework can grow incredibly powerful.

All these things easily lead to the framework's architecture becoming rather complex. However, there is much danger in complexity.

First, a complex architecture is difficult to build, maintain, understand, and use. Excessively complex frameworks can therefore create a challenge for designers and users alike, perhaps more of a challenge than they can meet.

Second, complexity, in the context of framework development, often means additional abstractions that are often reflected by generic algorithms and data structures. Genericity, however, is often the enemy of efficiency, and too many abstractions can easily lead you into efficiency problems you cannot resolve.

Third, the more complex the framework is, the longer it takes to build it. If the framework is supposed to "solve all problems on earth," it won't be available anytime soon.

This, however, is unacceptable. You don't have much time to build your framework. The teams that build the applications rely on the framework, and they won't be willing to wait for you. A specification of the framework has to be available when the other teams start designing, and a first version of the framework has to be completed before the other teams start coding. If you don't manage to get the framework ready in time, this will turn out extremely expensive for the entire project.

And no, you cannot rescue the situation by adding more people to the framework team when the deadline is approaching and the schedule is getting tight. We all know that adding people to a late project makes the project later [Brooks1995].

Solution

Design your framework to be small and to focus on a few concrete tasks.

Try to limit the scope of the framework to what is truly necessary:

- Focus on a small number of core concepts. Avoid too much abstraction. A framework that embodies too much abstraction tries to do too much and is likely to end up doing little.
- Perhaps your framework can be a framelet [Pree+1999]. A framelet is a very small framework that defines an abstract architecture that's not for an entire application but for some well-defined part of an application. It follows the "don't call us, we'll call you" principle, but only for that part of the application.

Much in the vein of agile development methods [Ambler2002][Cockburn2002], try "the simplest thing that could possibly work" [Beck2000]. Go with a small solution that works, rather than with a complex solution that promises more than it can deliver.

Examples

The Data Access Layer Framework When assembling the requirements for the database access framework, the team had to work hard to exclude any application logic from the framework. Many applications were going to use the framework's versioning features, but all had different ideas on how to use them. The team had to fight to avoid a versioning system that could be used in many different modes. Had the framework team agreed to include all the features that some of the other teams desired, it would have blown up the framework to incredible proportions.

In order to not overcomplicate the framework, the team also had to restrict the mapping of logical entities onto physical tables. Only simple mappings were possible; advanced techniques such as overflow tables had to be left to the applications. Generating the database access functions would otherwise have become unmanageable.

Nonetheless, the data access layer framework did suffer from too much complexity. The composition of business objects from smaller entities turned out to be extremely complicated. The team managed to implement it properly in the end, but only after much unexpected work. In addition, this complexity made the framework more difficult to use than had been intended. Looking back, the team felt the framework should have done without this mechanism, which required much effort and did little good.

To summarize, the team was successful in keeping the framework simple in many instances but felt they should have made simplicity an even more important issue.

The Web Portal Framework Providing a portal infrastructure is a concrete task, and certainly one that can be addressed by a framework. Like so many other frameworks, however, this Web portal framework suffered from becoming too powerful and, as a consequence, too complex. For instance, parameters need to be passed between the Web browser and the applications running on the mainframe, and the framework includes a mechanism that manages parameter passing in an entirely generic way. It's powerful, but it's difficult to understand and rather inefficient. A simpler mechanism would have been better.

But there is a success story as well. The framework wasn't only supposed to provide an infrastructure for the Web portal; it was also supposed to provide one for integrating Web services (which don't expect results from the applications to be represented as HTML). The framework team managed to meet these requirements in a simple and elegant way. The framework features a layered architecture: presentation issues are dealt with only in the servlet engine, while use-case management is concentrated on the application server layer. Web services can use the application server layer exactly as the portal does and simply not make use of any HTML generation.

Discussion

Reducing a framework's complexity is a strategy generally approved of in the literature. For instance, Art Jolin recommends that frameworks be simple and modeless [Jolin1999]. And in our particular context—time constraints along with framework and application development happening simultaneously—keeping the framework simple was crucial.

Simplicity also contributes to longevity. For instance, Brian Foote and Joseph Yoder propose THE SELFISH CLASS [Foote+1998]—a class that represents an artifact that reliably solves a useful problem in a direct and comprehensible fashion. In a similar vein, a framework that focuses on a set of core abstractions has a relatively high ability to evolve gracefully and therefore stands the best chance for longevity [Foote+1996].

In order to keep the framework simple as the project goes on, you must be careful with change requests that other teams might have, as they might introduce more, and unwanted, complexity. As a general rule of thumb, only accept MULTIPLE CHANGE REQUESTS (8)—change requests that are made by several, at least two, application development teams.

And of course a SKILLED TEAM (3) is critical to keeping the framework simple. Framework development requires a team that is aware of the problems complexity brings and that is able to see the beauty of simplicity.

3. Skilled Team

Problem

How can you make sure that the framework is designed in a clear and consistent way?

Context

You are going to develop a framework in your project. You now have to assemble a team that can perform the requirements analysis for the framework, define its architecture, and ultimately design and implement it.

Forces

Framework development is hard. But when framework and application development are strongly interwoven, things are even harder.

First, you have only a little time to develop your framework, because the other teams of the project are already counting on it. If your team is either inexperienced with framework development or with the application domain, there is no chance for success. The team would need too much time for familiarization.

Second, the framework team will have to take care of not only framework development and maintenance, but coaching as well. A crucial part of the framework team's job is to explain to the users how to work with the framework properly, which represents quite some time and effort.

Third, several applications are going to use the framework, so it is likely that many different parties would like to put their stake in it. You have to expect many different requirements and requests to influence the framework's design and scope.

And while such influences can give the framework designers valuable input, there is also the danger that the framework evolves into more and more variations. If different parties are free to request functionality as they see fit, the framework is likely not only to become complex, but to also split into inconsistent variations.

Solution

One team of skilled individuals must take care of framework design and development.

A skilled team is important in every development project, but it is crucial to framework development and even more crucial if framework and application development happen simultaneously.

Check the following, carefully and in detail, when assembling the framework team:

- The team members must have the necessary skills to design a framework. Experience with framework design and managing abstraction are required.
- The team members must adopt a strategy to keep the framework reasonably simple and to avoid unnecessary complexity.
- The team must have the experience and the skill to ensure that the framework's scope stays on target.
- The team must have the communication skills that are necessary to collaborate closely with the framework users and to incorporate feedback the users may have into the design.
- At least some team members must be familiar with the application domain.

In order to collaborate smoothly, the team should be large enough to accomplish the task, but no larger. Interestingly, a small team is sometimes more effective than a large team.

Examples

The Data Access Layer Framework Five people were involved in building the data access layer framework, although some only with a small percentage of their time. A stable core team of two people worked on the framework full-time for more than a year. When it came to introducing the framework into the applications, these two people were faced with the problem of doing three things at a time: maintaining the just-released version, preparing a new version, and coaching. Though they eventually managed this, there were some delays.

In retrospect, it became clear that a team of three or four people would have been necessary for timely releases and appropriate user support. A still larger team, however, wouldn't have done any good; the fact that only a handful of people designed the framework added much to its consistency.

The Web Portal Framework The good news here was that framework designers brought the necessary skills; they had developed frameworks before on other projects. The team size was also appropriate; about five people worked on the Web portal framework, which allowed for efficient teamwork.

However, a number of ad hoc corrections were made to the framework by people outside the framework team. This led to some irritation, since at some point responsibility for the framework wasn't clearly defined. After a major refactoring, the framework's consistency was reestablished, and responsibility for the framework was reassigned to the framework team.

Discussion

If too many people work on the framework, it's hard to develop one consistent architectural vision, which is what your framework needs. A small team can envision THE BEAUTY OF SIMPLICITY (2).

Likewise, a small team can take responsibility for how the framework evolves by making sure only MULTIPLE CHANGE REQUESTS (8) are accepted.

In his generative development-process pattern language, Jim Coplien explains that it is important to SIZE THE ORGANIZATION [Coplien1995] and to SIZE THE SCHEDULE [Coplien1995] when building a software development organization in general. The importance of having the right number of people, as well as the right people from the start, is even more true for building frameworks since the additional level of abstraction makes adding people to the project even more difficult.

4. Pilot Applications

Problem

How can you detail the requirements for your framework?

Context

Work on the framework has begun. You're in the process of defining the framework's functionality. At the same time, some application development teams have already kicked off, while others perhaps haven't started yet.

Forces

There is some functionality that several of the applications-to-be will have in common. They will use it in slightly different ways, but they will share a set of common abstractions. However, none of these applications have been built so far.

You're locked in some kind of chicken-and-egg problem: The framework team needs to know quite a bit about the applications to be able to define the framework's functionality, while the application development teams rely on the framework to build the applications. It's hard to come up with the requirements for the framework in such a context.

Still, you have to find the right abstractions for your framework.

What makes things even more difficult is that different application development teams will place many, possibly conflicting, requirements on your framework. You'll have to prioritize those requirements in order to achieve a consistent framework design. If you try to please everybody, the framework will become very complex and might ultimately become a failure.

Solution

Discuss the framework's functionality with several pilot applications that are going to use the framework.

The pilot applications will use the first versions of the framework as soon as they are released. This may well include versions that feature only a partial functionality or that are otherwise still constrained in their usage. In a way, the pilot application developers act as beta testers and provide valuable feedback to the framework team.

In most cases, two pilot applications are appropriate. One pilot application alone might not be representative, and perhaps you cannot say whether a required function is crucial or just nice to have. On the other hand, three or more pilots might simply become difficult to handle. Two pilot applications still seem manageable, and it's unlikely that important requirements go unnoticed.

Keep in mind the following when choosing pilot applications:

- The pilot applications must be fairly typical of others that might use the framework.
- The pilot applications should be significant, so that the framework team keeps in close touch with some of the framework's premier users.
- The pilot applications must be applications that are being built relatively early in the time frame of the overall project.

Collaborating with the teams who work on the pilot applications will increase the knowledge exchange in both directions; you'll get feedback on how good your framework is, and the other teams will learn how to use it. Pilot applications will also force you to adopt a policy of early delivery, which is a well-established strategy for project risk reduction [Cockburn1998].

Unfortunately, pilot users can get the impression that they're doing your work when they use the framework at a very early stage, when its functionality is still incomplete and has a few bugs. Be aware of this, and make clear to the pilot users that in return they have the chance to influence the system they'll have to use.

Examples

The Data Access Layer Framework Among the new systems, the health insurance system was a very typical one. The framework team had many discussions with the team that built this system. These discussions particularly helped shape the understanding of two-dimensional versioning of application data—versioning that makes a difference between when a change becomes effective and when it becomes known. It's a subtle topic, and it was quite significant for the requirements analysis and for the framework design in the earliest stage of the project. The framework didn't feel on safe ground, though, until they got into detailed discussions with the team who built the new customer system. The new customer system had slightly different requirements on application data versioning. Both systems complemented each other well as far as architectural requirements were concerned.

The Web Portal Framework The life insurance system and customer system served as pilot applications for the Web portal framework. This was clearly a good choice, since these applications were typical for the portal's usage. For instance, a typical use case is that of a bank assistant who looks up a customer in the customer system and recommends certain life insurance products to back up a bank credit. This typical use case involves exactly the two applications that were chosen as pilot applications. The framework designers received a lot of input from collaborating with the application developers.

Discussion

Collaborating with the pilot users is a kind of FRAMEWORK USER INVOLVEMENT (7), but it's actually more than that. Involving the framework users has the primary goal of achieving a better understanding of the framework among the users once the framework is released, whereas the knowledge exchange with the pilot users is bi-directional.

Prototyping is a strategy generally approved of in the literature on reusable software. Brian Foote and William Opdyke recommend to PROTOTYPE A FIRST-PASS DESIGN [Foote+1995] when the goal is to design software that is usable today and reusable tomorrow, as is certainly the case with frameworks that are developed with large-scale reuse in mind.

The importance of feedback from users is generally acknowledged. In his generative development-process pattern language, Jim Coplien stresses that it is important to ENGAGE CUSTOMERS [Coplien1995], in particular, for quality assurance, mainly during the analysis stage of a project but also during the design and implementation stages. Along similar lines, speaking of customer interaction, Linda Rising emphasizes that IT'S A RELATIONSHIP NOT A SALE [Rising2000]. Speaking openly with customers—the framework users in this case—will give you valuable feedback about your product.

The pilot applications are not only useful for finding out the requirements for the framework; they also form the precondition for setting up PILOT-BASED TESTS (6).

5. Small Objects

Problem

How can you increase the framework's flexibility while restricting its complexity?

Context

The overall scope and functionality for the framework is clear. You're now in the process of breaking the overall functionality down into smaller pieces, such as individual objects or functions that applications can use.

Forces

Maybe your framework offers an interface to the applications that use it. Maybe it offers abstract classes that the concrete applications have to implement. Either way, it provides a certain amount of functionality to the applications. This functionality is typically expressed by a set of objects or methods—or functions, depending on the underlying technology. This invites the question of how these artifacts should be designed.

There are essentially two opposite approaches you can take. You can choose either a larger number of less powerful objects or a smaller number of more powerful objects.

From the application programmer's viewpoint, both a smaller number of objects and simpler objects are desirable, because both make the framework easier to understand. But because you have to offer a certain amount of functionality, you cannot reduce both the number of objects and the individual objects' complexity. You have to choose one and sacrifice the other.

Which option should be preferred?

You have to keep in mind that different applications will probably use your framework in slightly different ways. Combinatorics tell us that a larger number of less complex objects can be combined in many more different ways than a small number of very powerful objects. This flexibility represents a clear advantage.

In addition, complex objects are generally difficult to understand and difficult to reuse. This is true especially for objects with huge interfaces and methods that require many parameters.

Solution

When breaking down functionality into individual objects, favor a larger number of less powerful objects over a smaller number of more powerful objects.

Applications can then combine several objects to obtain a behavior that is tailored to their specific needs. This policy offers several advantages:

- The objects the framework offers will be better understood.
- Smaller objects have a better chance of meeting the users' needs, since they are less specific to a certain context.
- A larger number of smaller, somewhat atomic, objects allows for more combinations, and hence for an increased configurability of the application.

The price you have to pay for this strategy is that you cannot minimize the number of objects, but as long as the objects are fairly easy to understand, this seems a reasonable price to pay.

Examples

The Data Access Layer Framework The data access layer framework allows the loading of business objects into its cache where they can be processed. Typically, an application loads a policy object and changes it, thereby also changing the policy's state, which can be active, under revision, or offered to customers.

What happens when a policy object, one that is already in the cache, is requested? Should it be updated? Should the version in the cache be used instead? Different applications had different requirements. Some applications even needed to define a priority among states; for instance, an active object should be replaced by an object under revision but not vice versa. The framework team refused to include such a logic into the framework's function for loading objects. Instead, they implemented two smaller functions: one that tells applications whether a certain object is already available in the cache, and another that loads objects. Applications can combine these functions to implement their specific logic.

Another example: The data access layer keeps track of which objects have been changed. At the end of a session, applications can commit all or some of the changes to the database. The team decided not to implement a complex function

Manolescu_book.fm Page 418 Thursday, March 30, 2006 4:30 PM

that saved all changed objects, but again decided to offer two functions: one that listed all changed objects, and one that saved individual objects to the database. Applications can combine these functions to implement their strategies of which changes should be committed to the database as they see fit.

The Web Portal Framework The Web portal framework allows the applications to define certain use cases that specify the order in which user requests are processed and mapped onto calls of the back-end systems. The framework team decided to let the use cases' objects consist of smaller entities—so-called "user-steps"—that the application developers could aggregate to full-fledged use cases according to their specific needs.

This solution turned out quite successfully. The definition of use cases was easy to understand, and the flexibility of the use case definitions made the concept useful for many applications.

Discussion

A framework should display THE BEAUTY OF SIMPLICITY (2). Less functionality is often better than more functionality. But at some point we know that a certain functionality is not debatable, but strictly necessary. This pattern deals with the question of how this functionality can be implemented in such a way that different applications can use it most easily.

The suggestion to have small objects is similar to Don Roberts' and Ralph Johnson's suggestion to build frameworks from FINE-GRAINED OBJECTS [Johnson+ 1998] and Brian Foote's and Joseph Yoder's recommendation to design objects with a LOW SURFACE-TO-VOLUME RATIO [Foote+1998], that is, objects with small external interfaces.

The benefit of using small objects is also related to the observation that small modules are more likely to be reusable, because smaller modules make fewer assumptions about the architectural structure of the overall system [Garlan+1995]; hence the risk of an architectural mismatch between components is reduced.

6. Pilot-Based Tests

Problem

How can you test the framework sufficiently and reliably?

Context

You are in the process of coding the framework. Tests are necessary to make sure that the framework works correctly and reliably.

Forces

Testing is an important aspect of quality assurance. Testing is particularly important when you build a framework, since bugs would quickly manifest in all applications that use the framework.

However, testing a framework is difficult [Fayad+1999]. A framework alone is just an abstract architecture, not something that can be executed. In order to test the framework, you need a sample application that uses the framework and so acts as a test driver.

Moreover, when you test software, you need test cases with sufficient coverage. Using just one application as a test driver is probably insufficient, because this one application might not use all the features the framework offers.

In addition, it can be difficult to find realistic test scenarios when there are no precursor applications that you could use.

Solution

Set up tests based on the pilot applications.

These tests can take different forms:

- Identify core components from the pilot applications that call the framework, and then use these components as test drivers for the framework.
- Find typical use cases from the pilot applications and maintain them as a test suite.
- Shape these test cases into regression tests that you can run before every release of a new version of your framework.

However, you can't rely on the pilot applications alone. You also have to include some exotic scenarios in your test suite, ones that take your framework to its limits and that can detect more unexpected bugs.

In addition, you need test cases that test the time, performance, and stability of your framework under load.

Examples

The Data Access Layer Framework The two-dimensional versioning of application data had to be tested with real-world examples. The first time the framework team performed such realistic tests was in a two-week workshop together with the team that developed the health insurance system. In this workshop, the framework team learned some subtle details about two-dimensional versioning that they had not yet implemented. The framework team was therefore able to do some fine-tuning at a relatively early stage. Overall, these tests were very successful, because the few changes that were necessary could quickly be made. These tests would not have been possible without the pilot users—the health insurance application team.

Moreover, the realistic examples that were used in the workshop represented typical use cases for the health insurance system. The framework team used these scenarios as a test suite for a series of future framework versions.

The customer system acted as the second pilot application. The framework team occasionally tested together with the customer system team, since this reduced the necessary testing effort for both teams. Everybody was able to fix problems very quickly, which was equally good for both teams.

The Web Portal Framework Tests of the Web portal framework were always performed using the pilot applications. It was extremely convenient to have two applications at hand that provided realistic test cases. New versions of the framework were only released after they had been approved by test teams that had checked whether the integration of the life insurance system and the customer system worked smoothly.

In addition to this, performance tests were carried out from time to time to check the portal's time performance.

Discussion

Testing is an activity where the PILOT APPLICATIONS (4) are particularly helpful. Finding real-world test scenarios would otherwise be very difficult.

Joining efforts with the users for testing is a particular kind of FRAMEWORK USER INVOLVEMENT (7) from which both the framework developers and users can benefit. The framework developers receive valuable test scenarios, while the framework users can run tests with the framework developers readily available for immediate bug fixing, if necessary.

7. Framework User Involvement

Problem

How can you make sure that members of the other teams will be able to use your framework when they build their applications?

Context

You have completed a first version of the framework. It's now the application developers' turn to use the framework in their applications.

Forces

Other teams depend on your framework in order to complete their applications—that is, to be successful in what they're doing. They want to know what the framework does and how it works. That's fair enough; you should let them know.

Empirical studies have shown that most people are willing to reuse software if it fits their needs [Frakes+1995]. You can therefore assume that the other teams are generally willing to use the framework as long as you can convince them that the framework offers the necessary functionality and that using the framework is easier than developing the functionality from scratch.

Moreover, frameworks often trade efficiency for flexibility, at least to some degree [Fayad+1999]. When efficiency is critical, applications built with the framework may need some fine-tuning. They may also have to replace certain generic mechanisms with more concrete and more efficient ones. In any case, users might need help using the framework or even customizing it a bit.

Ultimately, it's your goal that the other teams use the framework successfully. If they don't, the failure will be blamed on you, the framework team, rather than on them, the application team.

Solution

Involve the teams that use your framework.

You must show the users how they should use the framework. The users must get an understanding of the framework's feel, so that they understand what they can and what they cannot expect from the framework and how they can integrate it into their applications.

Possible actions include:

- Run common workshops. Explain the steps that users have to take when they build applications with the framework.
- If possible, provide tools that support the framework's instantiation process and demonstrate how to use these tools.
- Provide examples of how an application and your framework collaborate.
- If necessary, show the users how to optimize the applications they are building using the framework.
- Prepare tutorials and documentation and make them available early. Make sure the documentation directly addresses the framework users as its TARGET READERS [Rüping2003] and maintains a FOCUS ON LONG-TERM RELEVANCE

[Rüping2003]—things that application developers will need to know in the long term, when the framework team might not be available anymore on a day-to-day basis.

- Combine written documentation and interactive workshops with the application developers to establish an atmosphere in which team members can share information of different kinds and formats—an atmosphere that can be described as an INFORMATION MARKETPLACE [Rüping2003].

The drawback is that involving the users a lot costs a lot of time and will probably take place while the framework is still developed further. You must make sure that framework development doesn't grind to a halt while you're busy running workshops.

Examples

The Data Access Layer Framework After the release of the first version of the framework, the framework team had a two-week workshop together with the team that developed the health insurance system. The health insurance team wanted to know what they had let themselves in for—how they could use the framework. The framework team showed them and at the same time had the opportunity to fine-tune the two-dimensional versioning of application data, since it was tested with real-life examples for the first time.

At some point, the framework team learned that the commission system had special efficiency requirements. The commission system team had to define a sophisticated mapping of business objects onto database tables—more sophisticated than could be defined in the framework's meta information. Both teams discussed a way to extend the data access layer of the application with a special module that implemented the special mapping.

The framework team provided a usage document that explained the necessary steps that application developers had to take to configure the framework and to use it in their specific context. The document was helpful, especially in combination with workshops like the ones just mentioned, in which the application programmers were shown how to do what was written in the document.

The Web Portal Framework The project managed to integrate the life insurance system and the customer system into the Web portal, despite the portal's relative complexity. Crucial for this success was the fact that the framework team and the application development team had offices next door to one another and that they were able and willing to collaborate closely. Informal communication was no problem, and communication channels were fast. The framework developers were available all the time to answer questions from the application developers. Actually, the framework developers and the application developers felt they were one team, sharing the common goal of bringing the portal to life.

Discussion

Unlike the collaboration with the PILOT APPLICATIONS (4), this pattern doesn't put the emphasis on how the framework team can learn from the framework's users (although it's fine if they do). The focus here is to provide a service to the users and help them.

Involving the users and working jointly on their tasks is generally acknowledged as a successful strategy to use to achieve this goal. In particular, this is true of frameworks, due to the additional level of abstraction and the sometimes non-trivial instantiation process [Eckstein1999].

Moreover, listening to the users, running common workshops, and so forth, helps to BUILD TRUST [Rising2000]. Trust is important because the users will view your framework as a third-party component; they will only be successful building their application if the framework works as it is supposed to.

When you explain how to use the framework, using design patterns is often useful, since they describe typical ways in which application programs can be put together [Johnson1997]. If you consider developing a tool that helps users build applications, keep in mind that a complicated mechanism is probably not justified. However, a simple script, perhaps based on object-oriented scripting languages, might save a lot of work [Ousterhout1999].

8. Multiple Change Requests

Problem

How can you prevent the framework from growing too complex as a consequence of change requests?

Context

Several applications use your framework. Application developers approach you with requests for additional functionality.

Forces

Regardless of how much functionality you have already included in your framework, some people will always ask you to include more. After all, if you add functions to the framework, the members of the other teams won't have to implement these functions themselves.

But if you accept all change requests, the framework might end up overloaded with functionality. Worse yet, different users might come up with conflicting change requests, and if you accept all of them, you put the framework's consistent architecture at risk.

You absolutely must keep the framework simple. In particular, you should avoid building the framework to run in different modes [Jolin1999]. So what should you do?

If only one application is interested in the additional functionality, that application's team can implement the desired functions on a concrete level at a much lower cost than would result if you implement them on an abstract level.

However, if several applications need additional functionality, it's probably useful to include that functionality—for all the reasons that justify a framework in the first place.

Solution

Accept change requests only if several teams will use the additional functionality. What the word *several* means will depend on the concrete situation. If a framework is used by just three or four applications, the fact that a change request is supported by two of them can be evidence enough to show that the change request is justified. If there are more applications that use the framework, the threshold might be higher. Either way, you must make sure that you add functionality only if it is of general use, not just useful in rare cases.

The following guidelines are helpful for dealing with change requests:

- Be active. Once you have received a change request, it's your job to figure out if it could be useful for more than just one team.
- Help users of the framework to add application-specific functions when their change request is rejected and they need to find an individual solution to solve their problem.

When a change request is accepted, make sure that it doesn't invalidate the framework's design. Apply refactoring techniques if necessary [Fowler1999].

Examples

The Data Access Layer Framework The data access layer framework normally allows committing changes to the database only at the end of a session. After a while, it turned out that both the workflow system and the printing system required an exception to this rule; certain changes had to be visible on the database immediately. Given the fact that two applications requested the change, the framework team decided to offer an additional function that commits changes to the database immediately as long as these changes are atomic and consistent.

The customer system needed special search functions that allowed searching for a person with an arbitrary combination of name, phone number, address, and other data. The data access layer framework never included such arbitrary queries, since they would have been very complex to implement and they might have easily ruined the system's time performance. Because no other team needed such queries, the framework team decided not to extend the framework but to show the customer system team how to extend their concrete application with the necessary functionality.

At some point, the framework team received several requests from different teams for extending the two-dimensional versioning. It turned out that what those teams needed could already be expressed. The desired extensions would have made things a little more comfortable for the other teams. However, everybody requested different comfort functions that, if implemented together, would have overcomplicated the framework. The framework team thus declined the change requests.

The Web Portal Framework Of all the change requests concerning the Web portal, the first thing the project had to decide was whether they addressed the framework or any of the applications. The rule of thumb was this: Only if both applications would benefit would the change request be justified. Only a change request made by both applications was thus a candidate that the framework team would consider. This policy avoided having the applications' functionality intrude into the framework.

Discussion

We know that building a framework is justified only when there is CONCRETE EVIDENCE FOR REUSE (1). This pattern is in sync with the principle that functionality should be added to a framework only if at least two applications can use it. It can be acceptable to bend the rule of three a little if a technically plausible change request is made by two applications, but if the requested functionality is useful only for one individual application, it certainly doesn't belong in the framework.

Conclusions

I'd like to conclude this chapter by considering the patterns and the core principles expressed by them from a slightly different perspective.

Over the last couple of years there have been many successful frameworks in the world of open source software. Examples include, but aren't limited to, frameworks that have emerged from the Apache Software Foundation [www.apache.org], such as Struts and Cocoon. Let's take a brief look at the characteristics of these frameworks and see if we can find evidence for why they have been (and still are) so successful.

- The frameworks keep a clear focus on their core tasks. You can say what these framework do in just a few words. Struts is a framework for the implementation of the model view controller pattern. Cocoon is a framework for XML-processing organized in a pipeline style. None of these frameworks tries to solve every problem on earth. Instead, the developers worked hard to retain the frameworks' scope.
- The frameworks address areas with a concrete potential for reuse. Web development is central to today's world of software, and you can find the aspects that the frameworks address in many applications worldwide.
- The frameworks are relatively easy to use. This doesn't mean they're trivial to use, but the frameworks aim to avoid unnecessary complexity and keep an eye on a straightforward usage process.
- There is a strong collaboration between the framework developers and the framework users. These frameworks haven't been developed in an ivory tower. We speak of the open source community—the word *community* alone suggests that there is a massive exchange of information involved. It's this community that gives the framework developers much useful feedback on what they do and what they plan to do.

Keeping a clear focus, aiming for straightforward solutions, avoiding complexity whenever possible—these appear to be key factors in many success stories on frameworks. I think it's important to understand that to achieve these things, a certain frame of mind is required to be shared among the project team members that gives preference to smaller and practical solutions over higher, but unachievable, goals. In addition, a strong collaboration between framework developers and users seems to be an equally important ingredient to successful framework development.

You can find all this advice running like an undercurrent through the patterns presented in this chapter and in the solutions they suggest. In this sense, the patterns should help you with your decision if you're considering building a framework and should provide you with much useful experience in running your framework projects successfully.

Acknowledgments

First, I'd like to thank the people who shared their ideas on framework development with me throughout the last couple of years. Special thanks go to the colleagues of sd&m software design & management AG, Germany, with whom I was happy to collaborate on the two projects described in this chapter.

Sending a paper to a pattern conference always gives you a huge amount of valuable feedback. This chapter has gone through the process twice. Neil Harrison was the shepherd for an earlier version at PLoP 2000, and Dragos Manolescu shepherded the current version for EuroPLoP 2003. Both provided useful comments and suggestions, and they helped me improve the chapter. Thanks also go to the participants of the PLoP 2000 and EuroPLoP 2003 workshops in which this chapter was discussed.

References

[www.apache.org] Apache Software Foundation. *www.apache.org*.

[Ambler2002] S. Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. New York: John Wiley & Sons, 2002.

[Beck2000] K. Beck. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley, 2000.

[Brooks1995] F. P. Brooks. *The Mythical Man-Month*. Reading, MA: Addison Wesley, 1995.

[Brugali+1997] D. Brugali, G. Menga, and A. Aarsten. "The Framework Life Span." In *Communications of the ACM*, Vol. 40, No. 10. ACM Press, October 1997.

[Cockburn1998] A. Cockburn. *Surviving Object-Oriented Projects—A Manager's Guide*. Reading, MA: Addison-Wesley, 1998.

[Cockburn2002] A. Cockburn. *Agile Software Development*. Boston: Addison-Wesley, 2002.

[Coplien1995] J. O. Coplien. "A Generative Development-Process Pattern Language." In J. Coplien and D. Schmidt (eds.), *Pattern Languages of Program Design.* Reading, MA: Addison-Wesley, 1995.

[Eckstein1999] J. Eckstein. "Empowering Framework Users." In M. Fayad, R. Johnson, and D. Schmidt (eds.), *Building Application Frameworks—Object-Oriented Foundations of Framework Design*. New York: John Wiley & Sons, 1999.

[Fayad+1999] M. E. Fayad, R. E. Johnson, and D. C. Schmidt. "Application Frameworks." In M. Fayad, R. Johnson, and D. Schmidt (eds.), *Building Application Frameworks—Object-Oriented Foundations of Framework Design.* New York: John Wiley & Sons, 1999.

[Foote+1995] B. Foote and W. F. Opdyke. "Lifecycle and Refactoring Patterns That Support Evolution and Reuse." In J. Coplien and D. Schmidt (eds.), *Pattern Languages of Program Design.* Reading, MA: Addison-Wesley, 1995.

[Foote+1996] B. Foote and J. Yoder. "Evolution, Architecture, and Metamorphosis." In J. Vlissides, J. Coplien, and N. Kerth (eds.), *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996.

[Foote+1998] B. Foote and J. Yoder. "The Selfish Class." In R. Martin, D. Riehle, and F. Buschmann (eds.), *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998.

[Fowler1999] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Frakes+1995] W. B. Frakes and C. J. Fox. "Sixteen Questions About Reuse." In *Communications of the ACM*, Vol. 38, No. 6. ACM Press, June 1995.

[Garlan+1995] D. Garlan, R. Allen, and J. Ockerbloom. "Architectural Mismatch, or Why it's Hard to Build Systems out of Existing Parts." In *Proceedings of the International Conference on Software Engineering, ICSE 17*. ACM Press, 1995.

[Jacobsen+1997] I. Jacobsen, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press, 1997.

[Johnson1997] R. E. Johnson. "Frameworks = (Components + Patterns)." In *Communications of the ACM*, Vol. 40, No. 10. ACM Press, October 1997.

[Johnson+1998] R. Johnson and D. Roberts. "Patterns for Evolving Frameworks." In R. Martin, D. Riehle, and F. Buschmann (eds.), *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1998.

[Jolin1999] A. Jolin. "Usability and Framework Design." In M. Fayad, R. Johnson, and D. Schmidt (eds.), *Building Application Frameworks—Object-Oriented Foundations of Framework Design.* New York: John Wiley & Sons, 1999.

[Ousterhout1999] J. K. Ousterhout. "Scripting: Higher Level Programming for the 21st Century." In *IEEE Computer*, Vol. 32, No. 3, March 1999.

[Pree+1999] W. Pree and K. Koskimies. "Framelets—Small is Beautiful." In M. Fayad, R. Johnson, and D. Schmidt (eds.), *Building Application Frame-*

works—Object-Oriented Foundations of Framework Design. New York: John Wiley & Sons, 1999.

[Rising2000] L. Rising. “Customer Interaction Patterns.” In N. Harrison, B. Foote, and H. Rohnert (eds.), *Pattern Languages of Program Design 4*, Boston: Addison-Wesley, 2000.

[Rüping2003] A. Rüping. *Agile Documentation—A Pattern Guide to Producing Lightweight Documents for Software Projects*. New York: John Wiley & Sons, 2003.

[Tracz1995] W. Tracz. *Confessions of a Used Program Salesman*. Reading, MA: Addison-Wesley, 1995.