Building Frameworks and Applications Simultanously

Andreas Rüping

sd&m software design & management AG

Thomas-Dehler-Straße 27

D-81737 München, Germany

e-mail: rueping@acm.org

Introduction

The decision to build a framework is often made after having built a number of similar applications successfully. Maybe a company has been selling similar systems to various customers for many years; building a framework can then reduce the effort and the time needed to build more applications. A framework evolves: ideally, the evolution process starts with the experiences gained with previous applications; the framework starts as a white-box framework and matures into a black-box framework as it is being used [2][11].

Still, this isn't always possible. Imagine the following scenario:

You're starting a large project with several teams building various applications. One team has to provide all kinds of shared functionality — modules that many applications will need, but which should be developed only once. Sometimes such modules are rather simple: a module for computing time and date, for instance. But sometimes they aren't so simple: many applications may need database access, but a general module for database access isn't trivial. Moreover, the more complicated shared modules often vary slightly from application to application.

To this end, the idea of building a framework for the functionality shared across applications springs to mind. This framework should offer functionality that many of the modules can use which will be built in the course of the project.

However, you cannot wait with the design of such a framework until you have built a number of applications — the framework would be available too late to be used in the project. The framework must be built in parallel to the applications that are going to use it.

Designing a framework in such a context is hard. You cannot draw on the experiences gained with previous applications. If you want to be successful, you have to take care to keep the framework simple, and you have to collaborate closely with the teams who will use the framework.

This paper presents a pattern language that helps the framework developer in such a situation. Most of the patterns are specific to building a framework and its applications simultanously. Some patterns are variations of patterns that apply to framework development in general; however, their focus is shifted towards the special context.

Guidelines for the Readers

This paper is organised as follows. We start by introducing a running example. This example describes a framework for database access and is taken from a large project in the insurance industry.

The actual pattern language follows. It consists of seven patterns mined from this project. Each pattern includes a problem statement, a discussion of forces, and a solution. Each pattern is explained with the running example and is related to other patterns. The problem section and the first paragraph of the solution section of each pattern form so-called pattlets — thumbnails (printed in boldface) that give you a quick overview.

We conclude with a discussion of known uses.

Overview

The following diagram gives an overview of the pattern language. Patterns that are connected influence each other.

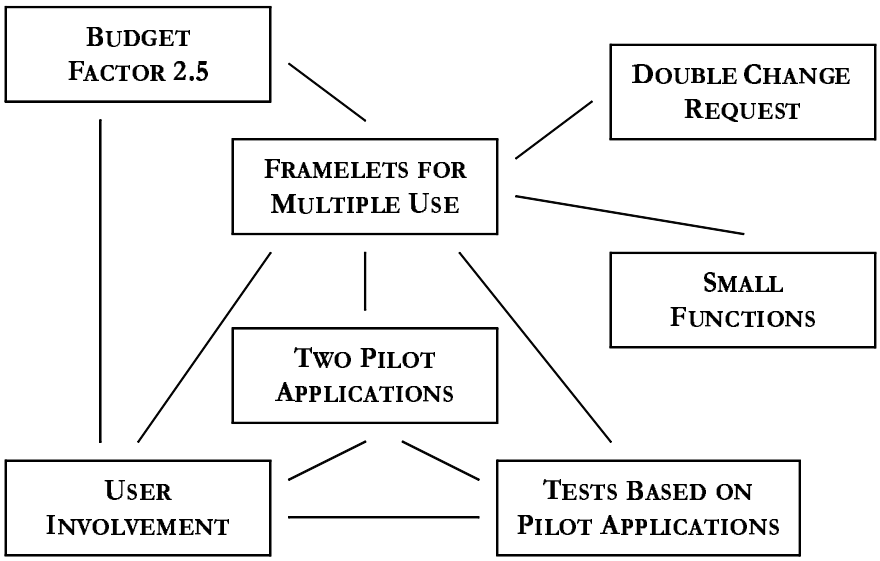

Figure 1 Overview of the pattern language

Running Example

The Project

A German insurance company faced the problem of having a large number of old legacy applications which didn't work together well. The company felt it was time for a change; they decided to build several new systems, including the following:

- a policy system for health insurance
- a policy system for life insurance
- a policy system for property insurance
- a party system that stores information about the persons involved in insurance contracts, either as insured persons, premium payers, or others
- a payment system that deals with the payment of both premiums and benefits
- a commission system
- a workflow system
- a printing system that prints all kinds of insurance documents.

In 1998, a large project was set up. Mixed teams from the insurance company and various software companies, ranging from 8 to 40 people, were formed to build the new systems.

One special team was given "horizontal tasks": the specification, design, and coding of modules that many, perhaps all other teams could use. The motivation was to save time and costs, as well as to ensure a consistent architecture across the new systems.

The Framework

All systems process large amounts of data stored in a relational database system; therefore each system needs a database access layer, as described in Figure 2. The access layers ensure that the applications can work on objects defined in the application domain and don't have to deal with the objects' physical representations.

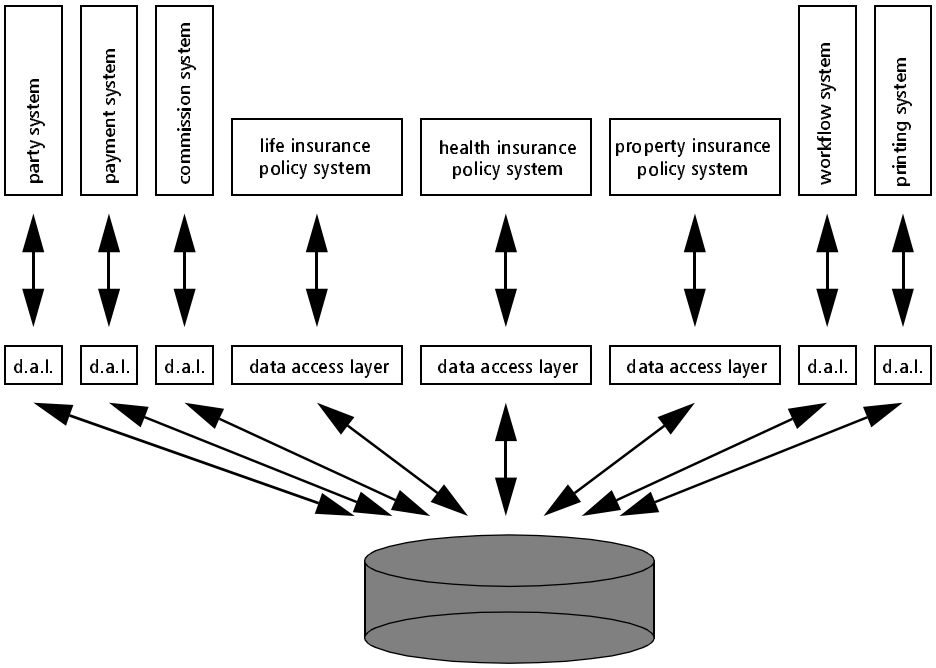

Figure 2 An insurance system

It soon became clear that providing database access would be among the project's horizontal tasks. Because the data access layers needed by the various new systems are architecturally similar but differ in many details, we decided to build a data access layer framework.

This framework features the following major hot spots:

- The framework provides views on domain-specific objects, such as policies and products. The mapping of domain-specific objects onto database tables differs from application to application. The framework abstracts over an application's data model.
- Insurance companies must keep versions of their application data. Access functions must be able to retrieve historical versions, for instance of a policy, from the database when necessary. However, the extent to which versioning is necessary differs from application to application.

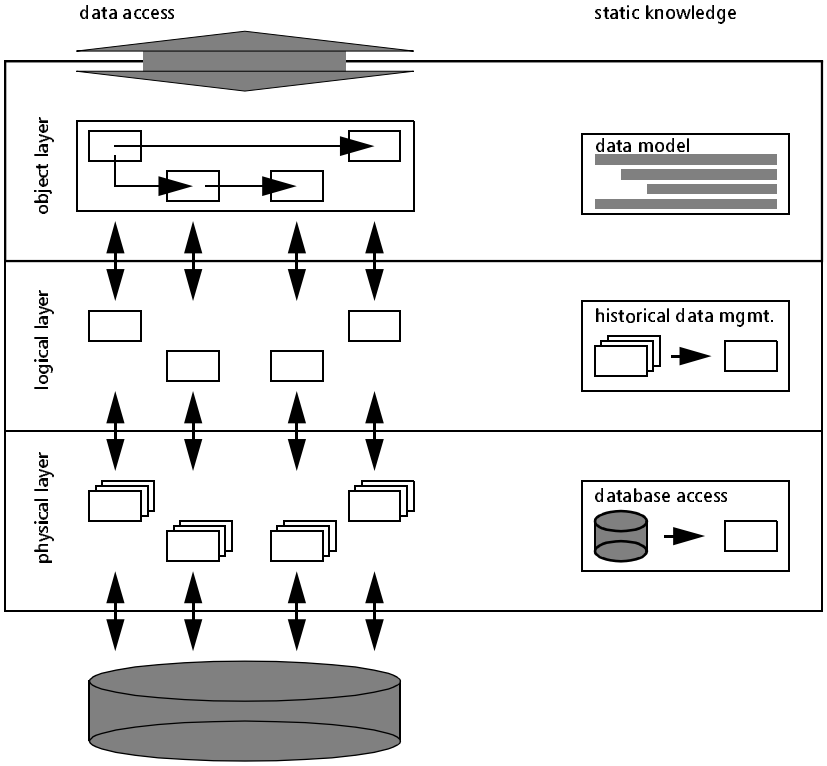

Figure 3 The architecture of a data access layer framework

The diagram in Figure 3 describes the architecture of the data access layer framework which itself consists of three layers.

The object layer represents the interface to the application program. It puts smaller entitites together to form complex, domain-specific objects. Application developers must provide the definition of the domain-specific objects through meta information. The framework includes an abstract class for the object layer. It also includes a mechanism that generates a concrete class for each domain-specific object from the meta information.

The logical layer manages the versioning of logical entities. The framework offers two abstract classes: one for entities that need versioning and one for entities that don't. Application developers define a concrete class for each logical entity and let it inherit from one of the abstract classes.

The physical layer stores and retrieves objects in the database. Application developers provide the mapping of logical entities onto database tables in the meta information. Again, the framework provides a mechanism that generates concrete classes. For each logical entity one concrete class is built which includes the actual access functions with SQL statements.

1 Framelets for Multiple Use

Problem

How can you justify building a framework simultanously to the applications that are going to use it?

Forces

There is some functionality that several of the applications-to-be will share. They will use it in slightly different ways, but there is a common abstraction. However, these applications don't yet exist, but will be developed in the same project in which the framework is going to be built.

In addition, the potential users of the framework might have competing interests in what the framework should do and how. Worse still, some of these conflicts might come up later, when the framework is already under development.

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 [10][18]. A framework needs to be used for three different applications before the break-even point is reached and the investment pays off.

Building a framework takes more time than building a normal application. A rough estimate is that framework development is two to three times longer, but the exact figure depends upon the framework's size and degree of abstraction.

You don't have much time to build your framework, though. A specification of the framework has to be available when the other teams start designing the applications that will use the framework. A first version of the framework has to be completed when the other teams start implementing, at the very latest.

Solution

Build a framework only if, first, you can make it quite simple and, second, you can expect it to be used by at least three, better four or five applications.

- The framework should be essentially a framelet [15]. A framelet is a very small framework that defines an abstract architecture not for an entire application, but for some well-defined part of an application. It follows the "don't call us, we call you" principle, but only for that part of the application.
- To keep the framework simple, focus on a small number of core concepts. Avoid too much abstraction. A framework with too much abstraction tries to do too much, and is likely to end up doing little.
- Three expected uses is a must, even for a small framework. Four or five expected uses is 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.

Example

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

When assembling the requirements for the database access framework, we had to work hard to exclude any application logic from the framework. Many applications use the framework's versioning features, but all had different ideas on how to use them. We had to fight to avoid a versioning system that could be used in many different modes. Had we 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 not to over-complicate the framework, we also had to restrict the mapping of logical entities onto physical tables. Only simple mappings are possible; advanced techniques such as overflow tables had to be left to the applications. Generating the database access functions would otherwise have become unmanageable.

Discussion

In their patterns for evolving frameworks, Don Roberts and Ralph Johnson claim that THREE EXAMPLES [11] should be developed before building a framework. While the context of that pattern is a scenario in which you're able to draw on the experience gained with building example applications, the context here is a project in which a framework has to be built simultanously to the applications that use it.

Juxtaposing the two scenarios, you can conclude that, if you can't draw on the experience of example applications, you can build a framework only if it is possible to keep the framework simple. It's a good rule of thumb for a framework to be simple and modeless anyway [13]. But under these specific conditions — time constraints, no precursor applications — keeping the framework simple is crucial.

In addition, because you cannot rely on experiences gained previously with THREE EXAMPLES [11], it's important to find TWO PILOT APPLICATIONS as soon as the framework project starts.

Both the number of expected uses and the relative simplicity of the framework are not only crucial for the success of a framework built in this particluar context, but they also form the basis for management issues concerning team and budget (BUDGET FACTOR 2.5).

2 Budget Factor 2.5

Problem

What budget do you calculate for your framework at the beginning of the project?

Forces

Building a framework takes more time than building a normal application. A rough estimate is that framework development requires a two to three times larger effort, but again, the exact figure depends upon the framework's size and degree of abstraction.

With the framework and the applications being built simultanously, you, as the framework team, sometimes have to do several things at a time: framework development, framework maintanance, and coaching. This is impossible if you don't have enough skilled people.

Despite the larger effort, you won't have much more time for building the framework than you would have had for an ordinary module. Many other teams will be waiting for your results; if you don't manage to get the framework ready in time, this will be extremely expensive.

You can't add more people to the framework team when the deadline is approaching and the schedule is getting tight. Adding people to a late project makes the project later [1].

Solution

Calculate about two and half times the budget you would need to build a single application. Find a team that brings the necessary skills for framework development. The team should be large enough from the start — about twice as many people as you would need to build a single application.

The extra budget is necessary for the following reasons:

- Finding the right abstractions isn't easy. Designing in the abstract needs more time and cross-checks between colleagues than working on a concrete level.
- The framework team must offer coaching to the other project teams who use the framework.

It can be difficult to argue that such a comparatively large budget is justified. You can argue along the following lines:

- If at least three applications will use the framework, developing the framework will still pay off, despite the larger budget.
- Coaching the users may cost time and effort, but it contributes significantly to the success of the overall project.

Example

It took us about 30 person months to complete the database access layer framework. Our estimation was 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 framework instantiation also takes time, three instances of reuse seems to be the break-even point in our example.

Five people were involved in building the data access layer framework, altough some only with a small percentage of their time. A stable core team of two people worked on the framework full-time. Developing the framework extended over about 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 we eventually managed this, we had some delays. We felt it would have taken a team of about four people to deliver the releases in time, and to support the other teams simultanously.

Discussion

In his generative development-process pattern language, Jim Coplien explains that it is important to SIZE THE ORGANISATION [4] and to SIZE THE SCHEDULE [4] when building a software development organisation 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.

The budget factor of 2.5 is closely related to a 'rule of three' from the literature on reuse, which says that the effort to build reusable software is about three times the effort to build non-reusable software [10][18]. The exact figures in the literature vary. A budget factor of 2.5 is fine, keeping in mind that we are focussing on small frameworks (FRAMELETS FOR MULTIPLE USE). The following estimates relate framework development, application development (without a framework), and framework instantiation (building a concrete application with a framework):

framework development = 2.5 * application development

3 * application development = framework development + 3 * framework instantiation

framework instantiation = 1/6 * application development

The increased budget is a precondition for coaching activities. It is therefore essential for the framework's success since without reasonable USER INVOLVEMENT and coaching, failure would be likely.

3 Two Pilot Applications

Problem

How can you find out the requirements for your framework?

Forces

There is some functionality that several of the applications-to-be will probably have in common. They will use it in slightly different ways, but there is a common abstraction.

However, none of these applications have been built so far. Moreover, you can't wait until a few applications have been built. You must perform a requirements analysis for the framework at the same time as the other teams perform the requirements analysis for the actual applications. It's hard to come up with the requirements ahead of time.

You have to find the right abstractions for your framework.

The other teams will place many, possibly conflicting requirements on your framework. If you try to please everybody, the framework will become very complex, and probably will ultimately fail.

Solution

Find two pilot applications that will use your framework.

- The pilot applications must be fairly typical.
- The pilot applications should be rather important, so as to keep in close touch with some of the prominent users of the framework.
- 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 a policy of early delivery, which is well-established strategy for project risk reduction [3].

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

The fact that there are two pilot applications will help you understand how important certain requirements are. With two pilot users it's easier to tell whether a required function is crucial or just nice to have.

On the other hand, more than two pilot applications can become unmanageable and will probably do more harm than good. You can't do everything at a time.

Example

Among the new systems, the health insurance system is a very typical one. We had many discussions with the team that built this system. These discussions particularly helped shape our understanding of two-dimensional versioning of application data — versioning that differentiates between when a change becomes effective and when it gets known. It's a subtle topic and it was quite significant for our requirements analysis and for our design in a very early stage of the project.

Not until we also got into detailed discussions with the team who built the new party system did we feel we were on safe ground, though. The new party system has slightly different requirements on application data versioning. Both systems complemented each other well as far as architectural requirements are concerned.

Discussion

Collaborating with the pilot users is a kind of USER INVOLVEMENT, but it's actually more than that. USER INVOLVEMENT 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.

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 [4] 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 [16]. 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 TESTS BASED ON PILOT APPLICATIONS.

4 Small Functions

Problem

How can you break down the functionality of your framework into interface functions?

Forces

Your framework is a framelet that covers a well-defined part of an application. Although the framework follows the "don't call us, we call you" principle for that part of the application, it has to be integrated with the rest of the application. It therefore has to offer an interface to the main event loop of the complete application.

A framework with a relatively small number of functions is more easily understood. A framework with relatively simple functions is more easily understood.

However, you need to put a certain amount of functionality into your framework. You can choose either a larger number of less powerful functions or a smaller number of more powerful functions.

Different applications will probably call the functions of your framework in slightly different ways.

Solution

Favour a larger number of less powerful functions over a smaller number of more powerful functions.

- The functions the framework offers will be better understood.
- Smaller functions 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, functions allows for more combinations, and hence for an increased configurability on the application's side.

The price you have to pay for this strategy is that you cannot minimize the number of functions in the framework's interface.

Example

The data access layer framework allows 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 should be loaded which is already in the cache? Should it be updated? Should the version in the cache be used instead? Different applications have different requirements. Some applications even need to define a priority among states; for instance, an active object should be overridden by an object under revision but not vice versa. We refused to include such a logic into the framework. Rather we implemented two 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. We decided not to implement a complex function that saves all changed objects, but, again, decided to offer two functions: one that lists all changed objects, and one that saves individual object to the database. Applications can combine these functions to implement their strategy of which changes should be committed to the database as they see fit.

Discussion

The first pattern to this collection of patterns, FRAMELETS FOR MULTIPLE USE, says that a framework for the functionality shared by several applications of a large project should be relatively simple and that functionality should be included if it is really necessary. In contrast to that pattern, this pattern assumes that a certain functionality is not debatable but strictly necessary, and deals with the question of how it can be implemented in such a way that different applications can use it most easily.

The suggestion to have small functions is similar to Don Roberts' and Ralph Johnson's suggestion to build frameworks from FINE-GRAINED OBJECTS [11]. It 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 [9]; hence the risk of an architectural mismatch between components is reduced.

5 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?

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 [8]. You can therefore assume that the other teams are generally willing to use the framework, provided you can convince them that the framework offers the necessary functionality and that using the framework is easier than developing the functionality from scratch.

Frameworks often trade efficiency for flexibility, at least to some degree [6]. When efficiency is critical, applications built with the framework may need some fine-tuning. Users might need help with this.

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.
- Offer tests of how an application and your framework collaborate.
- If necessary, show the users how to optimize the applications they are building using the framework.
- Make tutorials and documentation of the framework available early.

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 since you're busy running workshops.

Example

After the release of the first version of the framework, we had a two-week workshop together with the health insurance system team. They wanted to know what they had let themselves in for — how they could use our framework. We 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 we learned that the commision system had special efficiency requirements. The commision system team had to define a sophisticated mapping of business objects onto database tables — more sophisticated than can be defined in the meta information of our framework. We discussed a way to extend the data access layer of their application with a special module that implements the mapping they need.

Building a concrete data access layer doesn't require programming. Instead, application programmers replace generic parameters by actual parameters and choose superclasses from the framework to inherit from (for instance templates including or excluding versioning). This is a tedious and error-prone process that involves a lot of copy and paste. To make working with the framework easier, we provided a script that generates a concrete access layer from the application's meta information. A tutorial explains how to use this script, but we also demonstrated the automated instantiation process to the other teams.

Discussion

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

Involving the users and working jointly on their tasks is generally acknowledged as a successful strategy 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 [5].

Moreover, listening to the users, running common workshops, etc. helps to BUILD TRUST [16]. Trust is important since 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, design patterns are often useful since they describe typical ways in which application programs can be put together [12]. 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 [14].

Involving the users through coaching or workshops requires a budget which is sometimes underestimated at first. Calculating with a BUDGET FACTOR 2.5 is a precondition to allow for the necessary coaching activities.

6 Tests Based on Pilot Applications

Problem

How can you test the framework sufficiently and reliably?

Forces

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

However, testing a framework is difficult [6]. Normally, when you test software, you need test cases with sufficient coverage. Because of the framework's inherent abstraction, test case coverage for a framework is much harder to achieve.

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

Because your framework is a framelet, it covers only a part of an application and therefore cannot be tested alone. You need applications that use the framework to act as test drivers.

Solution

Set up regression tests based on the pilot applications.

In more detail, this means:

- Identify core components from the pilot applications that call the framework, and 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 only. 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.

Example

The two-dimensional versioning of application data included in our framework had to be tested with real-world examples. The first time we performed such realistic tests was in a two-week workshop together with the health insurance system team . In this workshop, we learned some subtle details about two-dimensional versioning that we had not yet implemented. We were therefore able to fine-tune our framework in a relatively early stage.

Moreover, the realistic examples that we used in the workshop represented typical use cases for the health insurance system. We could therefore use these scenarios as a test suite for the future versions of our framework.

The party system acted as the second pilot application for our framework. We occasionaly tested together with the party system team, since this decreased the necessary testing effort for both them and us. We were able to fix problems very quickly, which was equally good for both teams.

Discussion

Testing is an activity for that the TWO PILOT APPLICATIONS 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 USER INVOLVEMENT from which both the framework developers and the users can profit. The framework developers receive valuable test scenarios, while the users can run tests with the framework developers readily available for immediate bug fixing if necessary.

7 Double Change Request

Problem

How can you make sure your framework isn't overloaded with changes following requests for additional functionality?

Forces

Independent 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.

If only one application is interested in the additional functionality, that team can implement the desired functions on a concrete level at much lower costs than if you implemented 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.

Since many teams will use your framework for their applications, it's no surprise that they'll come up with different, maybe conflicting requests for more functionality.

However, you have to keep the framework simple. In particular, you should avoid that the framework can run in different modes. [13]

Solution

Accept change requests for additional functionality only if at least two teams will use the added functions.

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.
- Allow users of the framework to add application-specific functions when their change request is rejected.

When a change request is accepted, make sure that it doesn't invalidate the framework's design. Apply refactoring techniques if necessary [7].

Example

Normally the data access layer framework allows committing changes to the database only at the end of a session. Both the workflow system and the printing system required an exception to this rule; certain changes have to be visible on the database immediately. Changing the framework's session logic is rather difficult. We decided to offer an additional function that commits changes to the database immediately, provided these changes are atomic and consistent.

The party system needed special search functions that allow searching for a person with an arbitrary combination of name, phone number, address, and others. The data access layer framework doesn't include arbitrary queries since they would be very complex to implement in the abstract and they might easily ruin the system's time performance. Since no other team needed such queries we decided not to extend the framework, but to show the party system team how to extend their concrete application with the necessary functionality.

When we implemented the second release of our framework, we 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, different teams wanted different comfort functions which altogether would have over-complicated our framework. We declined the change requests.

Discussion

FRAMELETS FOR MULTIPLE USE requires that a framework for the functionality shared by several applications of a large project be used at least three times. The fact that the critical number is down to two for change requests is no contradiction: adding functionality is less complex than building a new framework.

Conclusions

The patterns presented in this collection were all mined from the insurance system project with which the patterns are explained. This project is representative in many ways: the need for new applications, framework and applications built simultanously, the time constraints, the involvement of several companies.

We made a lot of observations in this project about what worked and what didn't work, particularly as far as building frameworks is concerned [17]. We made some observations many times as the project continued and chose to describe these observations as patterns.

Other teams made similar observations. Our insurance project also saw the development of a printing system that receives textual building blocks from the applications and puts them together to form complete insurance documents. The printing system was designed as a framework that abstracts over the document structure, which differs depending on the type of insurance.

This relatively simple framework was also built simultanously to the applications that use it. The team reported that simplicity was crucial; they focussed on a small number of core concepts so that the complexity was still manageable, given the short time span in which the framework was built. The team also reported that joining efforts with the other teams for requirement analysis, use cases, and tests was an important prerequisite for the frameworks' success.

It appears that developing frameworks and applications simultanously is indeed possible, provided that the framework is quite small and that framework developers and users collaborate closely.

Acknowledgements

Thanks are due to all colleagues who worked on the data access layer framework, both inside and outside of sd&m software design & management, Munich, Germany. Particular thanks go out to Rudolf Simson, Christian Kölle, Achim Kugler, and Hans Zierer.

Thanks are due to Neil Harrison who, as the PLoP shepherd for this paper, made several suggestions for improvement. In particular, Neil suggested to contrast the patterns in this paper more with the related literature, and gave many helpful references. Thanks for making me think in directions I didn't think of at first.

Last but not least, I would like to thank the participants of the writers' workshop at PLoP 2000 in which this paper was discussed. Particular thanks go out to Paul Asman, Steve Berczuk, and Phillip Eskelin for providing many detailed comments.

References

[1] Frederick P. Brooks. *The Mythical Man-Month.* Addison Wesley, Anniversary Edition, 1995.

[2] Davide Brugali, Giuseppe Menga, Amund Aarsten. "The Framework Life Span", in *Communications of the ACM*, Vol. 40, No. 10. ACM Press, October 1997.

[3] Alistair Cockburn. *Surviving Object-Oriented Projects — A Manager's Guide.* Addison Wesley, 1998.

[4] James O. Coplien. "A Generative Development-Process Pattern Language", in J. Coplien, D. Schmidt (Eds.), *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[5] Jutta Eckstein. "Empowering Framework Users", in M. Fayad, R. Johnson, D. Schmidt (Eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design.* Wiley, 1999.

[6] Mohamed E. Fayad, Ralph E. Johnson, Douglas C. Schmidt. "Application Frameworks", in M. Fayad, R. Johnson, D. Schmidt (Eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design.* Wiley, 1999.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[8] William B. Frakes, Christopher J. Fox. "Sixteen Questions About Reuse", in *Communications of the ACM*, Vol. 38, No. 6. ACM Press, June 1995.

[9] David Garlan, Robert Allen, John 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.

[10] Ivar Jacobsen, Martin Griss, Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* ACM Press, 1997.

[11] Ralph Johnson, Don Roberts. "Evolving Frameworks", in: R. Martin, D. Riehle, F. Buschmann (Eds.), *Pattern Languages of Program Design*, Vol. 3, Addison-Wesley, 1998.

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

[13] Art Jolin. "Usability and Framework Design", in M. Fayad, R. Johnson, D. Schmidt (Eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design.* Wiley, 1999.

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

[15] Wolfgang Pree, Kai Koskimies. "Framelets — Small is Beautiful", in M. Fayad, R. Johnson, D. Schmidt (Eds.), *Building Application Frameworks — Object-Oriented Foundations of Framework Design.* Wiley, 1999.

[16] Linda Rising. "Customer Interaction Patterns", in: N. Harrison, B. Foote, H. Rohnert (Eds.), *Pattern Languages of Program Design*, Vol. 4, Addison-Wesley, 2000.

[17] Andreas Rüping. "Experiences with Object-Oriented Frameworks", in J. Eisenbiegler, M. Haug, B. Kölmel, E. Olson (Eds.), *Software Best Practices — Object-Oriented Concepts.* Springer, 2000 (to appear).

[18] Will Tracz. *Confessions of a Used Program Salesman.* Addison Wesley, 1995.