A Culture for
Small Frameworks

Patterns for Developing
Frameworks on the Fly

Andreas Rüping

sd&m software design & management AG
Lübecker Straße 1, D-22087 Hamburg, Germany

rueping@acm.org

Introduction

Object-oriented frameworks play an important role in many IT projects these days as they allow to reuse not only code, but also concepts and designs. Ideally, a framework evolves from experiences made while building several similar applications (Brugali Menga Aarsten 1997, Johnson Roberts 1998). A team that can rely on year-long experiences in an application domain stands the best chance for finding the right abstractions for a framework.

Sometimes, however, the need for a framework arises on a much shorter timescale. Think of a large development project in which several applications are going to be built. In its initial stage, the project might identify a certain functionality that several applications will require. The idea of building a framework springs to mind, so that this functionality can be reused wherever it is needed. It might therefore be an option to decide to build a framework dynamically during the project.

Such a context, however, is tricky. Building a framework requires generalisation from concrete problems. But as the framework and the applications will be built more or less simultaneously, such generalisation will be particularly difficult. In addition, the time frame for developing the framework is rather short, since the framework must become available during application development.

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

This paper presents a collection of patterns that address these questions, focusing on principles and strategies that offer a clear benefit when you are to develop a framework on the fly.

Project Background

I have observed the patterns included in this paper in several projects in which application development was the major goal, and in which a framework was built to facilitate application development. Experiences made in these projects were both positive and negative — in either case, the projects provided a rich basis for pattern mining.

I'll use two projects as running examples throughout this paper since these projects demonstrate the patterns particularly well. Let's therefore first take a look at these two projects.

The Data Access Layer Framework

An insurance company faced the problem of having a large number of old legacy systems which 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, a new customer system, a new payment system, a new commission system, and a new workflow system. All in all, this was a huge endeavour involving several teams, ranging from 8 to 40 people each.

Not all teams started work on their applications at exactly the same time, but the teams did have to work simultaneously in order to accomplish the overall goal of completing all applications by the time the company had in mind.

Early on, the project established a team that worked on "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, 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 project decided to build a framework that would provide access to a relational database, and that could be tailored to the individual needs of all applications.

Figure 1 illustrates the framework's architecture.[1] The framework itself consists of three layers.

- The object layer represents the interface to the various application programs. It provides views on domain-specific business objects such as policies and products, which are composed from smaller entities.

- The logical layer is responsible for the object versioning mechanisms that the framework includes, as is required for many insurance systems. These mechanisms make it possible to retrieve old versions, for instance the version of an insurance policy that was effective in the past.

- The physical layer stores and retrieves objects in the database.

---

1. The framework is represented by black elements while grey elements represent neighbouring systems.

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.

To use the framework, application developers must provide the mapping from domain-specific objects onto database tables. This mapping differs from application to application as the framework abstracts over an application's data model.

The extent to which versioning is required also differs from application to application. Application developers therefore have to configure the logical layer so that it implements object versioning to the extent that is necessary.
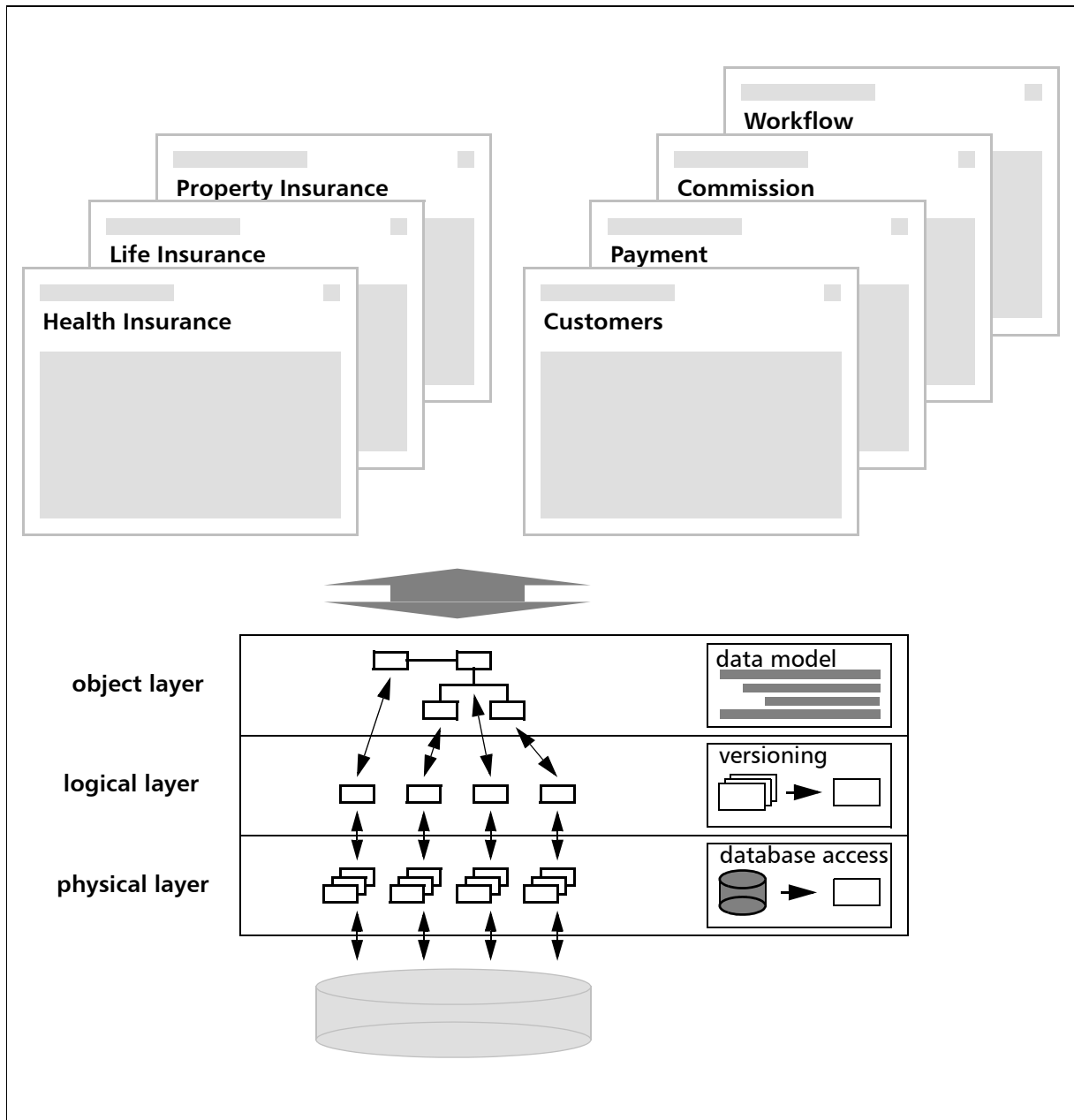

Figure 1   The architecture of the data access layer framework

The Web Portal Framework

The goal of this project was to develop a web portal for the financial industry. The portal includes both web content and applications, and is used by a bank to sell insurance products. The web content covers general information about the available products, while the applications provide access to different insurance systems which store and process contracts and customer information.

Several of these applications already existed when the plans for the portal were made, but a few still had to be developed. Besides application development, the project's major task was to integrate all applications into the portal. Mechanisms were needed to pass user input to the applications, and to transform the results into HTML so that they could be understood by the web browser.

The project came to the conclusion to develop a J2EE-based framework that should provide an infrastructure for all applications to be integrated. Building a framework was motivated by the fact that several applications had to be integrated into the portal, and that the necessary integration mechanisms could be reused.

Figure 2 shows the framework's layered architecture.[2] The framework itself extends over the web server and the application server, and accomplished the following tasks:

- On the web server layer, the framework accepts an http request from the web browser. The framework performs the necessary session handling, so that certain information is maintained over a series of http requests, and calls the application server.

- The application server handles the request in the context of its current use case, and calls an application on the mainframe machine to perform an actual operation. This can be a contract application or the customer application. The application server receives the response from that application, and manages its use cases accordingly, which essentially means that it knows which function of the application to call next time.

- Next, the application server passes the response on to the web server and the servlet engine.

- Finally, the web server and the servlet engine use JSPs and servlets to turn the response into HTML. They embed the results from the application into a presentation frame that all applications use to ensure a consistent layout, and pass the HTML to the web browser.

This framework does not help teams with building the application, but it helps them connect these application with the portal. What remains for the application developers to do — as far as integration is concerned — is to define the use cases that their application undergoes, so that the portal framework knows how, and in which order, to map client requests onto function calls. The definition of use cases requires a bit of programming, but the bulk of integration work is done by the framework and hidden from the application developers.

---

2. Again, black elements represent the framework while grey elements represent neighbouring systems.
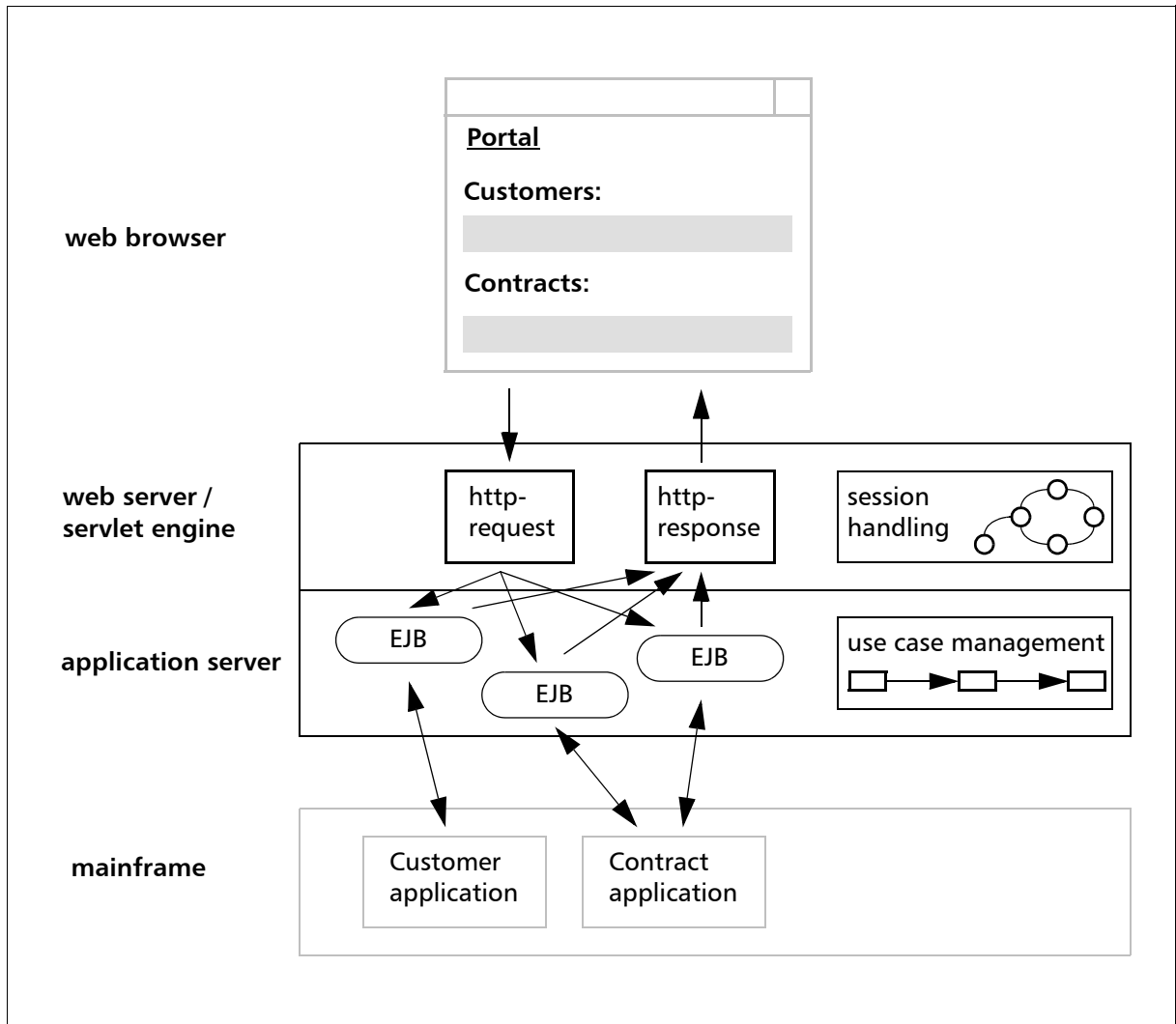
web browser

Portal

Customers:

Contracts:

web server /
servlet engine

http-request

http-response

session handling

application server

EJB

EJB

EJB

use case management

mainframe

Customer application

Contract application

Figure 2   The architecture of the web portal framework

Roadmap

Pattern Form The pattern form used in this paper begins with a problem section followed by a discussion of forces. Next, the solution shows how the forces can be resolved, and is explained with examples taken from the two projects mentioned above. Finally, the discussion section points out relationships between patterns as well as additional aspects.

The problem section and the first paragraph of the solution section are printed in bold face, and form a thumbnail sketch that give you a brief overview of each pattern.

References to other patterns are printed in small caps. If the referenced pattern is from this paper, the pattern name is followed by the *section number* in brackets. Otherwise, the pattern name is followed by a *reference* to the original source.

Overview Figure 3 gives an overview of the patterns presented in this paper and briefly sketches the relationships between the patterns.
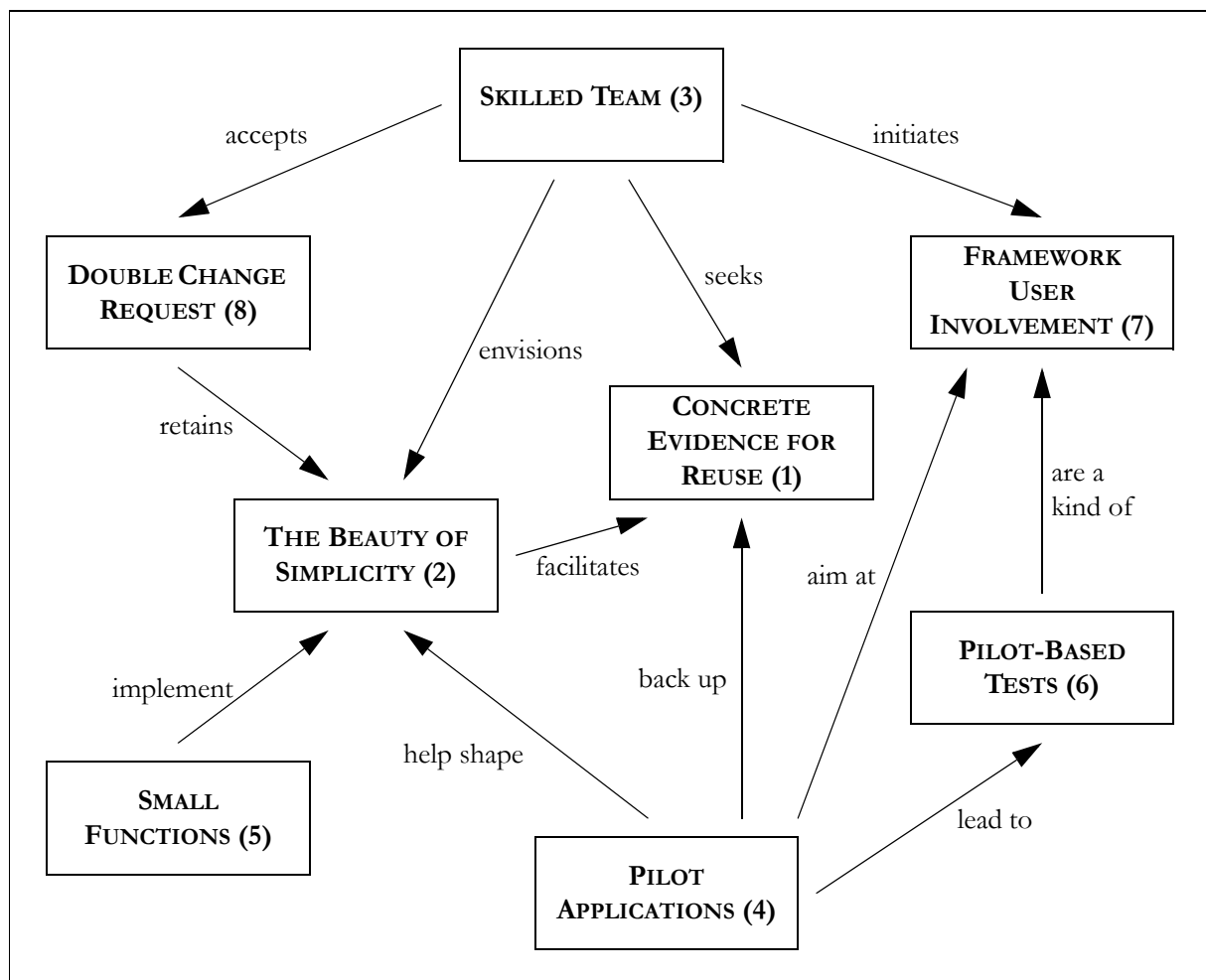
Figure 3   Overview of the pattern language

1 Concrete Evidence for Reuse

Problem   **How can you make sure that building a framework in a multi-application development project is justified?**

Forces   If you're working on a project which will see the development of several applications, it is sometimes tempting to design a framework: the framework could provide a set of common abstractions, and could serve as a blueprint for that functionality, which could then be reused several times. You could make this decision on the fly, 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 Griss Jonsson 1997, Tracz 1995). 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 finding a common denominator 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) developing a framework at the beginning of the project, because the development teams must know whether or not there will be a framework they can use. But there is an uncertainty behind whatever plan you'll have of which applications are going to be developed. This plan might change. New applications might be added, and others might be cancelled, including those that now serve as candidates for reuse, as far as the considerations for a framework are concerned.

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, the 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 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 seems to be the break-even point in our example.

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 recognised 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 Roberts 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 experiences.

Yet in our scenario, there are no three examples on which we could possibly rely. So is there anything you can do instead to make sure that our framework benefits from experiences with application development?

Yes, there is. As 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, the larger a framework is, the higher is the risk of wrong abstractions and hopes for reuse that won't materialise. Aim at THE BEAUTY OF SIMPLICITY (2).

2   The Beauty of Simplicity

Problem **How can you prevent your framework from becoming unmanageable?**

Forces When you discuss the possibility of building a framework, the application development teams will come up with requirements. After all, they are going to use the framework, so the framework must meet its purposes. If building a framework appears to be reasonable, application developers are sometimes inclined to add more and more requirements on the framework: the more functionality the framework offers, the more time and effort the application development teams can save.

Moreover, framework developers are sometimes in jeopardy of 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, to maintain, to understand, and to use. Excessively complex frameworks can impose a challenge on designers and users alike, and perhaps become more of a challenge than can be accomplished.

Second, complexity, in the context of framework development, often means additional abstractions, which 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 before too long.

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.

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 (Brooks 1995).

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 with too much abstraction tries to do too much, and is likely to end up doing little.

*   Perhaps your framework can be a framelet (Pree Koskimies 1999). 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.

Much in the vein of agile development methods (Ambler 2002, Cockburn 2002), try *the simplest thing that could possibly work* (Beck 2000). Rather go with a small solution that works than with a complex solution that promises more than it can keep.

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 not to over-complicate the framework, the team 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.

Nonetheless, the data access layer framework did suffer from too much complexity. The composition of business objects from smaller entities turned out a to be really 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 summarise, 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, the 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 it's 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, but also 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 use the application server layer exactly as the portal does and simply don't make use of 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 (Jolin 1999). And in our particular context — time constraints, no precursor applications — keeping the framework simple is crucial.

Keeping the framework simple isn't restricted to designing the framework, but extends to its evolution. 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 a DOUBLE CHANGE REQUEST (8) — a change request that is made by at least two of the framework users.

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

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

Framework development is hard. But when it comes to developing a framework on the fly, simultaneously to the applications that are going to use it, things are even harder.

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

Second, the framework team will have to take care not only of framework development and maintenance, but also of coaching. 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 an influence 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 also to split into inconsistent variations.

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

A skilled team is important in every development project, but is crucial to framework development, and even more critical in the particular context of developing a framework on the fly.

In more detail, check the following when assembling the framework team:

- The team members must have the necessary skills to design a framework. Experiences with framework design and managing abstraction are required.
- The team members must adopt the 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 not 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 fine; about five people worked on the web portal framework which allowed for efficient teamwork. However, a number of adhoc corrections were made to the framework by people outside the framework team. This lead to some irritation, as at some point, responsibility for the framework wasn't clearly defined.

After a major refactoring, the framework's consistency was re-established, and responsibility for the framework was re-assigned to 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 a DOUBLE CHANGE REQUEST (8) is accepted.

In his generative development-process pattern language, Jim Coplien explains that it is important to SIZE THE ORGANISATION (Coplien 1995) and to SIZE THE SCHEDULE (Coplien 1995) 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.

4   Pilot Applications

Problem   **How can you detail the requirements for your framework?**

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. 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.

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

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

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

The pilot applications will use versions of the framework as soon as they are released. In a way, they act as beta testers and provide valuable feedback to the framework team.

In most cases, two pilot applications are fine. One pilot 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 its unlikely that important requirements go unnoticed.

- 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 (Cockburn 1998).

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.

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 differentiates 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 a very early 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 are 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. These applications are absolutely typical for the portal's usage. Bank assistants can look up customers in the customer system, and they can recommend certain life insurance products to back up bank credits. All typical requirements on the framework were covered by these two applications, and 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.

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 (Coplien 1995) 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 (Rising 2000). 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 Functions

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

Forces If your framework is essentially a framelet, then it 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. This conjures up the question of how you should break down the interface into single functions the applications can call.

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

In order to make the framework easier to understand, both a smaller number of functions and simpler functions are desirable. But because you have to offer a certain amount of functionality, you have to choose one and sacrifice the other.

Which option should be preferred? In other words, which can easier be managed: the number of functions or their complexity?

You have to keep in mind that different applications will probably call the functions of your framework in slightly different ways. The option that makes it easier for application developers to use the framework in different ways therefore offers a clear advantage.

In addition, complex functions are generally difficult to understand and difficult to reuse.

Solution   **When breaking down interface functionality into individual functions, favour a larger number of less powerful functions over a smaller number of more powerful functions.**

Applications can then combine several functions to obtain a behaviour tailored to their specific needs. This policy offers several advantages:

- 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, but this seems a reasonable price to pay.

Examples   *The data access layer framework*

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 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. Rather they 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. The team 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.

The web portal framework

The web portal framework is called through standard http requests. Calling the framework is really easy, and was never the cause of any problems.

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 functions is similar to Don Roberts' and Ralph Johnson's suggestion to build frameworks from FINE-GRAINED OBJECTS (Johnson Roberts 1998). 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 (Garlan Allen Ockerbloom 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?**

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 Johnson Schmidt 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, as 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 on different forms:

- 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 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 health insurance system team. 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 in a relatively early stage. Overall, these tests were very successful, as the few changes that were necessary could quickly be made. Clearly these tests would not have been possible without the pilot users — the health insurance system 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 decreased 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 really 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 who had checked whether the integration of the life insurance system and the customer system worked smoothly.

In addition to this, performance test 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?**

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 Fox 1995). 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.

Moreover, frameworks often trade efficiency for flexibility, at least to some degree (Fayad Johnson Schmidt 1999). When efficiency is critical, applications built with the framework may need some fine-tuning. Or they may have to replace certain generic mechanisms by more concrete and more efficient ones. In any case, users might need help.

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 an example of how an application and your framework collaborate.
- If necessary, show the users how to optimise 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.

Examples *The data access layer framework*

After the release of the first version of the framework, the framework had a two-week workshop together with the health insurance system team. 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 their application with a special module that implemented the special mapping.

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, the framework provided a script that generates a concrete access layer from the application's meta information. A tutorial explains how to use this script, but the framework team also held which workshops in which they demonstrated the automated instantiation process to the other teams.

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 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 learning from the framework's users (although it's fine if you 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 achieve this goal. In particular this is true of frameworks, due to the additional level of abstraction and the sometimes non-trivial instantiation process (Eckstein 1999).

Moreover, listening to the users, running common workshops, etc. helps to BUILD TRUST (Rising 2000). 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 (Johnson 1997). 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 (Ousterhout 1999).

8 Double Change Request

Problem

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

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.

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 those, you put the framework's consistent architecture at risk.

Still, you have to keep the framework simple. In particular, you should avoid that the framework can run in different modes (Jolin 1999). So what should you do?

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.

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 (Fowler 1999).

Examples   *The data access layer framework*

Normally the data access layer framework 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, provided these changes are atomic and consistent.

The customer 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 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 which altogether would have over-complicated the framework. The framework team 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: only if both applications would benefit, the change request might be justified. Not that it was justified automatically, but a change request made by both applications was a candidate the framework team would consider. This policy avoided the application functionality to permeate into the framework.

Discussion   We know that building a framework is justified only when there is CONCRETE EVIDENCE FOR REUSE (1). This pattern is sync with that principle: functionality implemented following a change request can be reused only if at least two applications can use it.

A question that might come up here is why a *double* change request is considered fine, while for developing a framework *three* applications are a must. This seems to be a contradiction. However, a change request typically requires adding a piece of functionality or a little fine-tuning, but normally won't require finding new common abstractions between different applications. Change requests made to a framework typically pay off a bit more quickly than developing a framework from scratch. Change requests to an existing framework can therefore be justified at a reuse level that wouldn't allow for the development of a new framework.

Conclusions

It appears that developing a framework on the fly is indeed possible, provided that the framework is kept small and that framework developers and users collaborate closely.

Again, this is not easy to achieve. It requires a frame of mind or an attitude, to be shared among the project team, that gives preference to smaller and practical solutions over higher but unachievable goals.

And while this seems to be a good strategy in general, it is crucial when it comes to developing frameworks quickly on demand. You can be successful only if you can establish a culture for small frameworks.

Acknowledgements

References

Ambler 2002
Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process.* Wiley, 2002.

Beck 2000
Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.

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

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

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

Cockburn 2002
Alistair Cockburn. *Agile Software Development.* Addison Wesley, 2002.

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

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

Fayad Johnson Schmidt 1999
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.

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

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

Garlan Allen Ockerbloom 1995
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.

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

Johnson 1997

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

Johnson Roberts 1998

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

Jolin 1999

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.

Ousterhout 1999

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

Pree Koskimies 1999

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.

Rising 2000

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

Tracz 1995

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