eXtreme Frameworking

–

How to aim applications at evolving frameworks

Stefan Roock
University of Hamburg
Roock@informatik.uni-hamburg.de
http://swt-www.informatik.uni-hamburg.de/people/sr.html
Vogt-Kölln-Str. 30
D-22527 Hamburg

Abstract

eXtreme Programming (XP) is a modern and powerful approach to the development of critical and high innovative software. Framework development is always critical and highly innovative. Therefore XP should be suitable for framework development.

The XP techniques create often new versions of the software under development. Since application programs depend on frameworks the migration process of the application programs to the new framework version becomes crucial.

This paper discusses the versioning of frameworks and the migration of applications. First approaches to the problem are presented.

Motivation and Overview

„You'll never get it right the first time" – this is one of the central insights on which called eXtreme Programming (XP) is based. (cf. Kent Beck [Beck1999], Ward Cunningham [Cunnigham1995]). Especially pair-programming (cf. [Beck1999]), refactoring (cf. [Opdyke1992], [Fowler1999]) and aggressive unit test (cf. [Jeffries1999], [Fowler1999]) are suitable techniques to achieve dramatic improvements of code and design quality. These techniques are not the only XP techniques but they are the ones that influence the versioning of frameworks and migration of application programs most when XP is used for framework development.

We develop the JWAM framework (cf. [JWAM]) with XP techniques since the beginning of 1999 and we observed remarkable improvements in quality (understandibility, simplicity, correctness, robustness etc.) since then. We call the usage of XP techniques for framework development and application „eXtreme Frameworking" (XF). This paper is located in this setting of Java framework development with XP techniques.

During the JWAM development with XP techniques we observed major problems with the versioning of the framework and the migration of applications. In the following sections I first stress the importance of quality for framework development. The quality requirements demand the mentioned XP techniques. Then I describe the technical source of the versioning and migration problem – the open-closed principle. The following section faces the question, who may be affected by framework versioning. The discussion of possible solutions for the versioning and migration problem needs a solid conceptual base. Such a base is created by the section „Dimensions of versioning and migration". On this base a possible migration process with supporting tools is sketched.

Quality in framework development

Quality is crucial to framework development. The users of frameworks are application developers. They have to understand the architecture of the framework. Documentation of classes and relationships between them is neccessary, but sometimes not enough. The source code is the ultimative specification of a class and its operations. Knowledge of the concrete implementation may not be desireable but in practice it can be the last rescue to understand a class or operation ([Meyer1995]). Therefore it is often suitable to have the source code of the framework at hand.

Often software quality is separated into internal and external quality factors (cf. [Meyer1997]). External quality factors matter to the users of the software system, internal ones matter to the developers only. This separation vanishes for frameworks. Users of the framework – the application developers – should have access to the architecture and source code of the framework. Therefore there is nothing which can be called „internals“.

It seems obvious that the XP techniques match with framework development. But there are problems!

Framework development is only worth the efforts if the framework is used in a lot of application projects. These application projects create stability pressure on the framework developers. Modifications made to the framework may disturb the application projects, since the application programs have to be modified to work with the new framework versions. The XP techniques (especially refactoring) will modify the framework often. Therefore we need techniques to allow continuous refactoring of the framework while not disturbing the application development in an inacceptable way.

The source of the versioning problem

As indicated, an important aspect of framework development is the handling of versions. Since frameworks are used for the development of different software projects often located in different developer organizations, it is not always possible to synchronize the production of the framework with the production of the applications. Then, frameworks have their own problems and demands on the development process. What seems a small requirement from the application programmer’s viewpoint may call for a major reconstruction of a framework. Therefore frameworks evolve rather independently of their applications. But they have to fit with their applications and their further development.
Using frameworks which evolve over time leads to a major problem with the otherwise extremely useful open-closed principle introduced by Meyer (cf. [Meyer1997]). The open-closed principle means that in an object-oriented system the interface of a class can be fixed (i.e. closed) for re-use by other clients; at the same time the interface can be evolved (i.e. open) through subclassing for further development. This well-established principle has to be revisited when further developing software frameworks:
Applications using software frameworks usually have to subclass several classes of that (so-called white-box-) framework. If a framework evolves this might mean a change of the superclasses for its applications, thus invalidating the application programs. This problem is also known as the „fragile base class problem“ (cf. [Szyperski1997]).

Targets of Versioning

There are different groups which may be affected by a new version of a framework or component[1]:

- The *framework developers*.
- The *component developers* of components which are based on the modified framework or component.
- The *application developers* of an application system using the component or framework.
- The *end users* of the application system.

I focus on the groups which already worked with a previous version of the framework. Previous versions of the framework don't matter to new users of the framework, since no migration has to be done.

Framework Developers

It has to be ensured that all framework developers work on the same version of the framework. This can be achieved by a version control system like CVS, RCS etc.

Continuous integration may cause some problems for the framework developers. If the framework becomes quite large the integration may take a while since a lot of test cases have to be executed. If a lot of integrations a required per day, this overhead may be too much. In this case specialized integration tools are useful.

In the context of the JWAM framework development the integration takes 30 to 60 minutes each. This time is due to the fact that a lot of tests have to be executed. Until now not all tests run automatically. Especially tests of interactive parts have to be executed manually. We are now constructing a client-server integration tool which allows the integration of new or modified classes from the clients in a asynchronous way. All tests are executed by the integration server and the results are reported to the client. If some tests fail, the integration will be rejected by the integration server. To succeed with this tool we have to automate as much test cases as possible. Therefore we are currently extending the JWAM framework with special test widgets wich allow us to automate even the interactive tests.

Component Developers

Component developers differ from application developers since other developers (application developers and developers of higher components) depend on their work. Like applications components have to migrate to the new framework version. In a world of frameworks and framework based components the application developers depend not only on the framework but also on the components. Therefore they do not simply need the new version of the framework to migrate the application but also the migrated components. Therefore application developers need the new version of the components together with the new version of the framework.

Component developers need access to a new framework version before the application developers. This is the only chance for the component developers to provide migrated versions of the components in time.

---

[1] This paper doesn't contrast frameworks and components. In the context of this paper a framework is always a component framework and components are always based on a framework. An example of such a component framework is the JWAM framework (cf. [JWAM]).

This can only be ensured with a proper development process. A possible development process provides a framework with stable interfaces one or two months earlier to the component developers than to the application developers.

Application Developers

Application developers have to migrate application programs to new versions of the used framework and components. Therefore they neeed the new version of the framework and the components at the same time.

The application has to be migrated to the new framework version. If the application is large the migration may be a major effort.

Notice that not only the application programs have to be migrated but also the application developers. They have to learn the modifications of the new framework version: classes may have be renamed, operations may have been removed etc.

End Users

If the framework and components used by the application program are linked into the application system, framework and component versions don't matter to end users. If the framework and components are not linked into one program file (like Java class files, DLLs, etc.), the situation differs. Several programs are installed on the end user's computer. These may depend on different versions of the same framework or component. Therefore we need a naming mechanism which ensures the selection of the appropriate version. Including the version information in the framework and component names solves this problem (cf. [Szyperski1997] and section „Freeze Version") at a first glance. But the migration of persistent objects become much harder: Every evolved class has now a different name than its previous version.

Dimensions of versioning and migration

The versioning of frameworks and the migration of applications to new framework versions can be discussed along some dimensions (see Figure 1 and Figure 2).

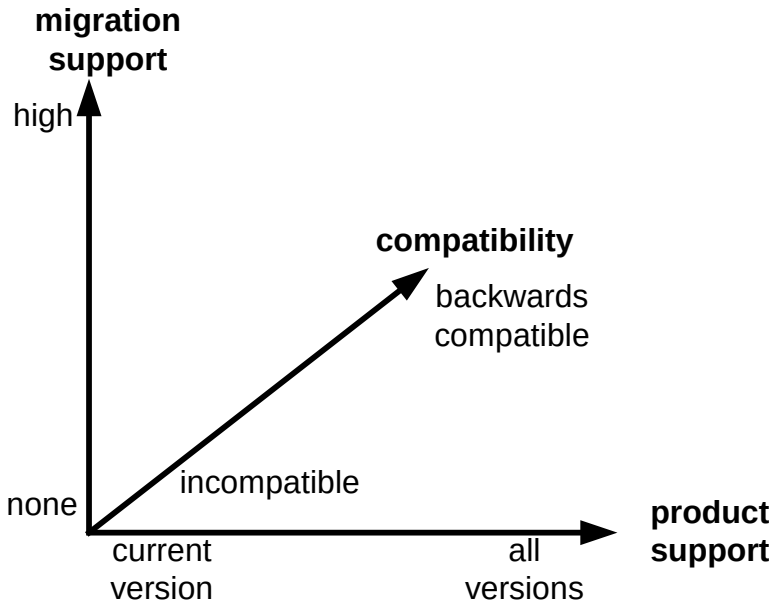

Figure 1 : Dimensions relevant to framework versioning

The three dimensions relevant to framework versioning are „migration support“, „compatibility“ and „product support“. The framework developers may choose to support the migration to new framework versions. This can be done in several ways: a history file describing what’s new and different, a guide through the new features of the framework, a tool supporting the migration, provision of man power for the migration process etc. The demand on migration support varies with the compatibility: Is the new framework version incompatible with the previous one or is it backwards compatible? Both dimensions influence the product support. If the migration to the new framework version is trivial the framework developers may choose to support only the current version of the framework. If they can’t expect the application developers to migrate immediatly they may choose to support more than one version or even all versions of the framework.

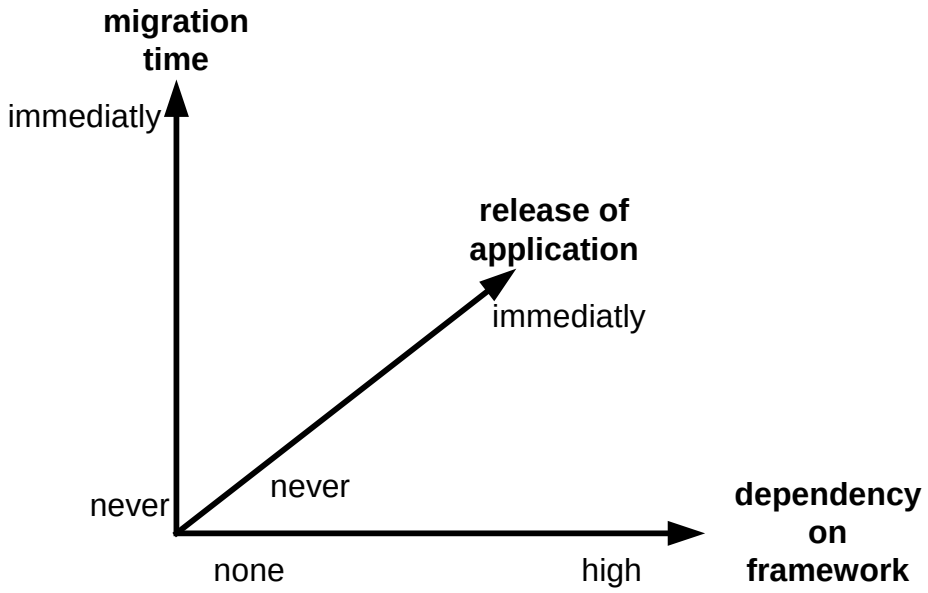

Figure 2: Dimensions relevant to application migration

The dimensions relevant to application migration are „migration time“, „release of application“ and „dependency on framework“. The application developers may choose to migrate immediatly to a new framework version or later. They may even choose to migrate never to the new framework version. The dependency on the framework may influence the choice of the migration time. If the dependency is low it is easier to migrate immediatly to a new framework version than it is if the dependency if high. To lower migration efforts application developers may therefore choose to lower the dependencies on the framework. Both dimensions influence the release of the application. If they migrate immediatly to the new framework version they may want to delay the release of the next application version until they have executed some tests with the application based on the new framework version.

In the following sections I describe some points of special interest in the discussed dimensions.

Freeze Version

A very simple strategy is to freeze versions. That means that on a conceptual level no modifications are made to an existing version of the framework. To support this strategy the version information is part of the components name. An example of such a strategy is Microsofts COM (cf. [Szyperski1997]). Another example is the „Java product versioning specification“ (cf. [Java1998]).

An interesting point is the aspect of error patches without affecting the interface. On a theoretical level such error patches are useful. On a practical level, error patches may cause problems. Clients of the components may have included work arounds and corrective functions to correct the errors of the component. Installation of an error patch will result in errors in the client. Therefore error patches to components are not allowed after relase in the „Java versioning specification“.

This strategy seems to solve the problem at first. On the other hand it is obvious that the component must evolve for error corrections and functional extensions. The „Freeze Version“ strategy ensures that running clients are not affected when a new version of the component is installed. This is suitable for end user's: If a system is installed together with a new framework version, the already installed programs don't change their behaviour.

But the application and component developers want to use corrections and functional extensions to the components. The „Freeze Version“ strategy doesn't support this case.

Backwards Compatibility

Backwards compatibility of often used for artifacts which are used by a really large number of clients. Prominent examples are chip design (Intel 8086 till 80486) and operating systems (MS-Windows operating systems 3.1/95/98). Within this strategy it is possible to extend the functionality of a component if the existing functionality is not modified. Backwards compatibility provides the possibility to use old and new client code with the same version of the component.

This approach has some drawbacks. With every new version of a component, more overhead is added to the component. New client code can use the new functionality of the new component version but has to accept the overhead. A component with much overhead is hard to maintain and extend. It is very hard to *ensure* backwards compatibility for frameworks due to possible side effects.

Deprecated Tags

In some programming languages (Java [ArnoldGosling1998], Eiffel [Meyer1997]) it is possible to mark interfaces, classes and operations as deprected (in Java with the deprecated meta tag; in Eiffel with the obsolete keyword). If a client uses a deprecated interface, class or operation, compiling and executing the program is possible but the compiler prints produces warnings referring to the client code which uses the deprecated entity. This way the application program may migrate to the new version of the framework in small steps.

If an operation should be deleted, it can be marked as deprecated and removed in a later version. If an operation should be renamed, a new operation with the new name can be introduced and the old one can be marked as deprecated.

This mechanism is powerful and easy to use, but it has problems, too. The first concerns the question, how long a deprecated entity should remain. On a theoretical level this question may be easy to answer: „one version“. On a practical level it isn't that easy. Very large applications may need more than one version to migrate to the new framework version. When we look at the Java Development Kit (JDK) we can detect a lot of operations which are deprecated since version 1.1 (actual version is 1.2).

The second problem is related to moving classes between packages or renaming packages. In this case whole packages with deprecated classes may exist. Since these deprecated classes are not in an inheritance relationship with the new classes, typing problems may occur.

The third deficiency is that deprecated tags do not cover all possible refactorings. Modifications to the pre- and postconditions of an operation or to the interactions of objects can not easily be covered with the deprecation mechanism.

Parallel Versions

This strategy provides three versions of a framework every time: The outdated version, the current and the future version (cf. [RoockWolfZüllighoven1998]). This strategy introduces a life cycle of a framework. This life cycle means that a framework version goes through three stages over time:

1. The current version
2. The outdated version
3. The coming version.

At each point in time, a framework version is in one of the above three stages. The *current version* is the version that should be used by 'active' application projects. It is supported and maintained but the interfaces do not change. Any application shipped to a customer has to be based on the current version of a framework.
Applications in operation may use the *outdated version* of a framework. The outdated version is still supported but no longer enhanced. Developers maintaining an application which is based on the outdated version know that they have to migrate to the current version.
The *coming version* is the next coming framework version. It has a well-defined interface and functionality which are published when a version is declared to be the new current one. Software projects may use the coming version as a specification from the start in order to access new interfaces or functionality of a framework. The prerequisite is that they can ship their application only after the coming version has become the current one. The version life cycle of application programs and framework is shown in Figure 3. In the example version 3 of the framework is coming in July 1998, version 2 is the current version and version 1 is outdated.

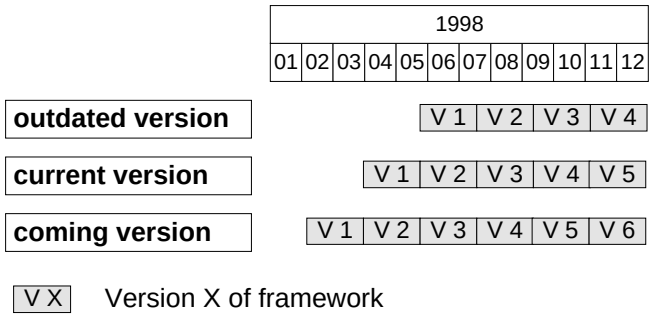

Figure 3: Example versions of a framework

Parallel versions require more efforts than a single version concept. The parallel version concept shares an important problem with deprecated tags: One version may not be enough to migrate all application programs.

Discussion of versioning and migration dimensions

None of the described dimensions can be discussed isolated from the others since the items in the dimensions are not fully independent from each other. From the XP viewpoint a combination of „Deprecated Tag" and „Parallel Versions" seems to be reasonable. As discussed this may cause big migration efforts for the application developers, especially if a lot of applications rely on the framework. Therefore high migration support is demanded. In the next two sections this migration support is discussed.

With this versioning concept in mind the application developers have the choice if they want to migrate to the new framework version immediatly or at a later time. They also have the possibility to release their new application version at a arbitrary time. Due to the fact that the framework evolution may still cause some migration effort, they may want to choose to lower the dependency on the framework.

The Migration Process

Now we have analyzed the target groups of new framework versions and possible techniques for handling the versioning problem. Based on this analysis I discuss first ideas of versioning a and migration process in this section.

Whenever a new version of a framework is released, the following steps are neccessary to upgrade the application software based on the framework:

1. Detect the modifications made to the framework since the last version.
2. Figure out, which modifications are neccessary to the application program.
3. Do the modifications to the application program.
4. Ensure that the behaviour of the application program is preserved.

These modifications to the application are crucial for the migration process but that is not enough. The application developers must „migrate“ to the new framework version as well. Application developers develop routine while dealing with the framework. Part of this routine may be invalidated through the new framework version: operations may have been deleted, new abstract operations have to be redefined to run the application properly, classes may have been renamed etc.

If the application developer wants to extend his application he has to know what is new and different in the new framework version to adapt his way of programming.

Therefore an additional step is neccessary:

5. Adapt the routines of the application developers.

Test classes are useful for steps 1,2 and especially 4. In addition to test classes documentation of the modifications is crucial to succeed with steps 1 and 2. Documentation is the key to step 5. Step 3 is somewhat special. In principle it is clear what to do if steps 1 and 2 were successful. Nevertheless the work required for step 3 may be awesome, error prone and costly due to the amount of source code which has to be modified in large projects. Tool support for step 3 is possible (see next section).

Tool Support

In this section I discuss some ideas about possible tool support for the versioning of frameworks and the migration of application programs.

We have seen that the migration of the application programs to a new framework version should be supported by appropriate tools. One central tool is the Modification-Detector. The Modification-Detector is able to detect design deltas between versions of source code. These design deltas contain the common refactorings (see [Fowler1999]) but are not restricted to them.

On the base of the detected design deltas a computer readable history is created by the Modification-Detector. The History-Browser reads the generated history and provides different views on the history. The History-Browser is also able to to create a human readable history.

The Migrator tool supports the migration of the application software. It gets the history and the application software as input and creates a new version of the application software as output. These modification to the application software can be done automatically for some framework modifications (like renaming of framework classes and operations). Other modifications (like adding new abstract operations) must be done manually by application developers. These manual modifications are also supported by the Migrator tool.

In contrast with the Smalltalk refactoring browser the described tools work offline: The modification of the framework is done at another time and location than the modification of the applications. First versions of the described tools covering a subset of all possible modifications are finished; the Migrator tool is under construction now. The completion of teh tools will be a major part of my future work.

Discussion and Future Work

I have discussed the general aspects of versioning frameworks and the migration of the application projects. I pointed out that not only the application software must be migrated but the application developers as well.

One important topic is tool support for the modification of the application software. I have sketched a small toolbox for this task.

Two aspects are in the focus of my future research in this area: The completion of the versioning and migration toolbox (VMT) and the migration of the application developers.

Acknowledgements

I’d like to thank my collegues at the University of Hamburg for their support of my work, namely: Heinz Züllighoven, Guido Gryczan, Henning Wolf and Martin Lippert.

Literature

[ArnoldGosling1998] Ken Arnold, James Gosling. The Java Programming Language. 2nd Edition. Addison Wesley. 1998.

[Beck1999] Kent Beck. Extreme Programming Explained. Embrace Change. Addison-Wesley. 1999.

[Cunningham1995] Ward Cunnigham. EPISODES: A Pattern Language of Competitive Development. In: Vlissides, Coplien, Kerth (Eds.). Pattern Languages of Program Design 2. Addison-Wesley. Reading Massachusetts. pp. 371-388. 1995.

[Fowler1999] Martin Fowler. Refactoring. Improving the Design of existing Code. Addison-Wesley. Reading Massachusetts. 1999.

[Java1998]. Java Product Versioning Specification. http://www.javasoft.com. February 10, 1998.

[Jeffries1999] Ronald E. Jeffries. eXtreme Testing. In: Software Testing & Quality Engineering. March/April 1999. pp. 23- 26.

[JUnit1999] Kent Beck, Erich Gamma. http://www.armaties.com/D/home/armaties/ftp/TestingFramework/Junit. 1999.

[JWAM] The JWAM framework. http://www.jwam.de. 1999.

[Meyer1995] Bertrand Meyer. Object Success. Prentice Hall. London. 1995.

[Meyer1997] Bertrand Meyer. Object-Oriented Software Construction. Second Edition. Prentice Hall. New Jersey. 1997.

[Opdyke1992] William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD Thesis. University of Illinois at Urbana-Champaign. 1992.

[RoockWolfZüllighoven1998] Stefan Roock, Henning Wolf, Heinz Züllighoven. Frameworking. In: Niels Jakob Buch, Jan Damsgaard, Lars Bo Eriksen, Jakob H. Iversen, Peter Axel Nielsen (Eds.): IRIS 21 "Information Systems Research in Collaboration with Industry", Proceedings of the 21st Information Systems Research Seminar in Scandinavia, 8 - 11 August 1998 at Sæby Søbald, Denmark. pp. 743-758. 1998.

[Szyperski1997] Clemens Szyperski. Component Software. Addison-Wesley. Harlow, England. 1997.