JWAM and XP
Using XP for framework development

Martin Lippert, Stefan Roock, Henning Wolf, Heinz Züllighoven
University of Hamburg, Computer Science Department
Software Engineering Group
&
APCON Workplace Solutions Company
{lippert, roock, wolf, zuelligh}@informatik.uni-hamburg.de

Abstract

We started using the XP techniques for designing and implementing the JWAM framework since the beginning of 1999. With the help of these techniques we succeeded in evolving the JWAM framework from a „student's project" into a real-life professional application framework which is used in several commercial applications today.

In this paper we report on our experiences with XP in general and with XP for framework development in particular for more than one year. The following sections describe how we use the XP techniques and how we have adapted them to our specific programming domain – the development of application frameworks for large-scale application software. This paper also discusses some of the problems encountered during XP and their potential solutions.

History of JWAM

JWAM is a Java framework supporting the development of large scale interactive software systems according to the tools & materials approach[1]. The foundation of the JWAM framework was laid in 1997 by research assistants and students of the Software Engineering Group at the University of Hamburg and it was a pure University project. We used it as a sandbox for gaining some experience with new concepts and with framework development in general. We also used it for teaching purposes.

In 1998 we felt that JWAM had the potential for professional software development. We thought it to be a solid technical base for large-scale software development giving support to developers with a proven design. Important steps to commercialize the framework were a redesign of parts of the framework and the explicit definition of a framework development process and its management. Early in 1999 we began to use XP techniques in a team of seven framework developers and redesigned parts of the framework. First, we used refactoring (cf. [Opdyke92], [Fowler99]), pair programming (cf. [Beck99]) and test classes ([Firesmith96], [Junit99]). Then we added the planning game and continuos integration (cf. [Beck99]).

The redesign of the framework had one major goal: simplifying the framework. With this goal in mind we refactored the framework and introduced a separation of the framework core from framework components based on this core. Before we began refactoring there was one framework compound with more than 600 classes. After refactoring we had a framework kernel with about 100 classes plus test classes. The rest of the original classes were divided into separate framework components or had become useless during the refactoring process.

[1] WAM is the German acronym for tools, automatons, materials. More information about WAM can be found in [RiehleZüllighoven95]. The framework can be downloaded from [JWAM].

Now, in January 2000 three application projects use JWAM. Two of these projects have already shipped operational client/server applications based on JWAM.

We use all of the XP techniques and we were quite successful in introducing them to our team. Of course we had to adapt some of the techniques to our situation. Currently, we further develop the JWAM framework in pairs only and nearly every framework class has a test class. If you want to know more about the JWAM framework take a look at [JWAM].

The Setting

The JWAM framework is both rooted in the university and has its commercial context. Within the university we use JWAM for teaching and as a sandbox for trying out new concepts. Within the commercial context we have founded the Apcon Workplace Solutions Ltd. The company uses JWAM for professional application development. This combination of an academic and industrial setting gives us the chance to use leading edge concepts in commercial projects very fast. On the other hand the requirements of the industrial projects trigger the research activities at the university.

Figure 1 shows the business use case for the development and usage of the JWAM framework. Note that the different actors may map to the same persons.

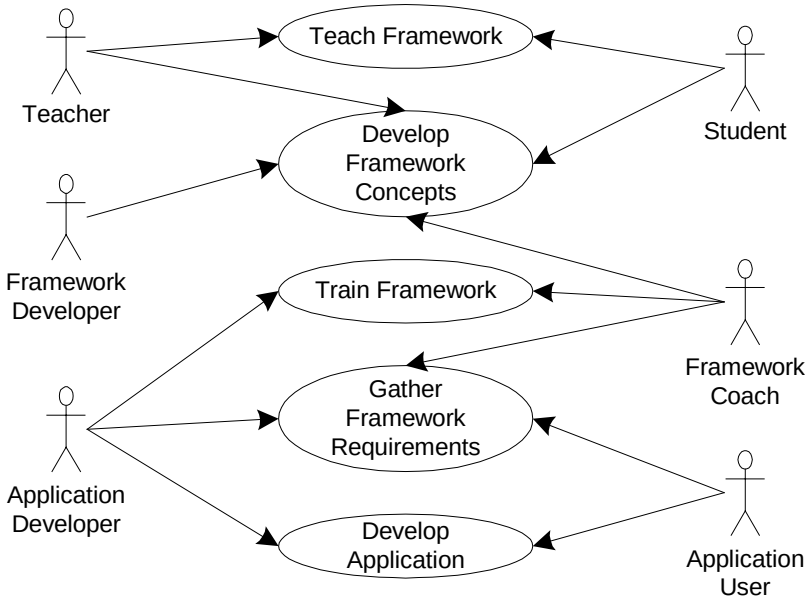

Figure 1: Business Use Case for JWAM development and usage

Framework developers may also develop applications, teach framework design or coach software teams. Application developers do nothing but application development, but may evolve over time into framework developers. Our students will both develop the framework and applications, but the don't teach or coach.

Lessons Learned

All of the XP techniques proved to be suitable for framework development. Of course we had to adapt some of the techniques and we learned that we were using others in a sub-optimal way at first. The following sections are structured like the book by Kent Beck ([Beck 99]) and explain our experiences using the XP techniques.

Planning Game

We use story cards for the planning game. Every framework developer may write story cards about new ideas he or she has or document requirements of the framework users (i.e. the application developers). In addition we, the framework developers, prioritize the story cards. We and not the users decide which one should be realized first. This usage of the story cards differs from what is described in [Beck 99], where the users, in our case the application developers using the framework, should rank the story cards. We've adapted the planning game to our needs in this way because we don't produce an individual piece of software. The framework is the technical base for many application developers spread over many companies and over different business domains. Therefore our users cannot coordinate themselves about the ranking of the story cards – we have to do it in line with our plans for the future of the framework.

Another adaptation of the planning game is the slightly different content of the story cards we write. In most cases these "in-house" story cards don't contain pure requirements like the original story cards described in [Beck 99]. We also use story cards to write down solutions to open problems and open requirements. The reason for this usage is the slightly different role of the story card writers. Many of the cards are written by a framework developer as a result of a discussion inside the team, with his/her pair (see the pair programming part) or with an application developer using the framework. The rest of the story cards are written by application developers while coding.

Since the JWAM framework isn't a pure in-house framework it is not that easy to integrate application developers into the framework development team. Nevertheless we feel that this is necessary. We try to solve the problem the other way around: Framework developers work as application developers part time. They pair with application developers and thus gain insight into their way of using the framework. In addition, expertise about the framework is spread into the application projects fast.

We have discovered that the stories are useful for planning but – in our case – they are not sufficient. We needed additional techniques to get a view of the overall picture – especially the interconnections between the story cards. If one story cannot be developed in the estimated period of time, it may be necessary to reschedule depending stories. In addition we needed to divide the bulk of story cards in handy portions and make our planning more transparent to our framework customers. Therefore we have enhanced the planning game by document types of the WAM approach (cf. [Züllighoven98]): *base lines* and *project stages*.

We use project stages and base lines for scheduling. A project stage defines which consistent and comprehensive components of the system should be available at what time. The project stages are an important document type for communicating with application developers. We use them to make the development progress more transparent to our customers. The project stages are the base for discussing the development plan and rescheduling it depending on users' needs. Often they are scheduled by the management of the software development team. Figure 2 shows an example of three project stages of our framework development process. We write down at what time we want to reach what goal and what we have to do to realize the aspired goal. Typically the project stages are scheduled backwards from the end to the beginning since most important external events (vacations, training programs, fixed dates for exhibitions, fixed dates for customer meetings and project meetings) and deadlines are fixed when projects are established.

Sub-goal	**Realization**	**When**
JWAM 1.5 can be delivered	Sample Application is running, all tests are fine	31.3.2000

New tool construction sub-framework designed	Review and refactor existing component, implement additional requirements requested by application developers	16.5.2000
Task oriented documentation ready for review by application developers	Improve task documentation for framework core, rewrite documentation for new tool construction	30.8.2000

Figure 2: Example project stages

Base lines are used to detailed planning within one project stage. They do not focus on dates but define what has to be done, who does it and who controls it in what way. In contrast to the project stages, base lines are scheduled from the beginning of the period to the end.

Within the base lines table (for example in figure 3) we write down, who is responsible for what base line and what it is for. The last column contains a remark how to check the result of the base line. The base lines table helps us to identify dependencies between different steps inside the framework development (see "What-for" column). The last three columns are the most important ones for us. The first column is not that important because everybody is able to do everything (like with story cards). But it is important for us to know how to check the results in order to get a good impression of the project's progress. It is an indicator for possible rescheduling between the base lines. It also helps us to sort the story cards which are on a finer-grained level.

Who	**does what with whom/what**	**What for**	**how to check**
Paul Bahr	Implementation of new tool construction	Preparation of tool construction review	email to team and new tool construction available to team
Paul Bahr, Richard Simon, Daniela Mohn	Review of new tool construction	Refactoring of new tool construction	Review documentation available to team
Daniela Mohn, Paul Bahr	Refactoring of new tool construction	Modification of existing tools based on old tool construction	Email to team and refactored tool construction package available to team
Andrea Kuhl, Richard Simon	Refactoring and adaption of existing tools based on old tool construction	Test of complete framework and all existing examples	All existing tools are running with new tool construction

Figure 3: Example of base lines

Pair Programming

Pair Programming has improved the quality (ease of use, simplicity, correctness) of the framework and has helped to spread the knowledge about the JWAM framework over the team.

The physical environment is crucial. We started with a conventional setting consisting of desks with fixed cabinets at their sides (see Figure 4)

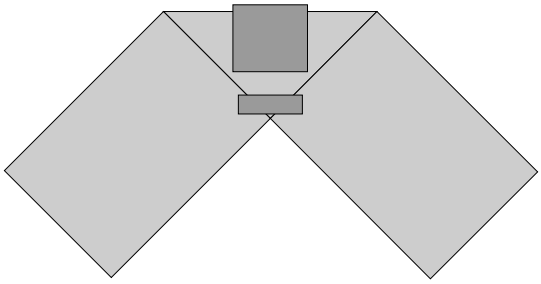

Figure 4: conventional environment

It was possible to program in pairs but the switching of roles was very uncomfortable: we had to get up and change our places. Therefore we switched roles only a few times per day. After a while we re-arranged the furniture and arrived at an environment which facilitates the switching of roles (see Figure 5)

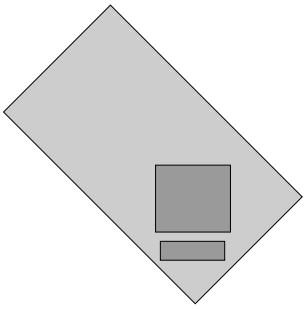

Figure 5: Re-arranged environment for better role switching

With the new environment it is much easier to switch roles and we do it frequently. But we think that a table with a circle at one end (see Figure 6) would be a further improvement. We will test it in the near future.

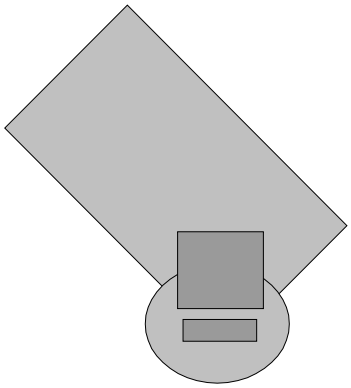

Figure 6: „Circle table“ for Pair-Programming

Typing speed is important when programming in pairs. Whenever you have to wait for your buddy due to his poor typing skills, you tend to become impatient. Then you will simply take

over the keyboard and you don’t want to switch again. This is a general lesson: If there are two developers with similar skills, pair programming is rather smooth. If one developer is much faster and has to explain a lot of things to the other one, he has to be a patient person. Nevertheless, pair programming is efficient for „training on the job“. After all, we can say that patience is worth it.

We have noticed that the role switching doesn’t happen as often as it should be according to [Beck 99]. This is no problem and we think we know the reason behind this: every story card is discussed by the framework team before it is realized by a pair. The discussion is not detailed enough to write the exact code for a card but it clarifies the design of the potential solution. Therefore both buddies have a clear vision of the solution and each one is able to write it down. As a consequence each buddy can write large sections of code without being interrupted or corrected by the partner. Keyboard switching in a pair occurs at “natural” breaks during programming. The term “natural” means interruptions intended by the buddy holding the keyboard.

Test Cases

With Pair Programming we have improved framework quality, with test cases we maintain it. Without the test cases a lot of the refactoring we did in the past would have been less smooth.

Another side effect of test cases is simplicity. Since it is much easier to test a simple interface, nobody will program a class with a complex interface if it isn’t necessary. Therefore new framework interfaces and classes are quite simple.

We were used to designing by contracts (with pre- and postconditions, cf. [Meyer97]) from the beginning of the JWAM development. We have found that test cases and contracts are complementary. Since the contracts are tested on executing an operation, we need not test primitive operations in detail. Often it is sufficient to simply call an operation. On the other hand the contract model has its weakness: A contract is based on the concept of abstract data types (ADT). ADTs not only have pre- and postconditions but also axioms. But only a very small part of the axioms can be expressed by invariants or postconditions because it is often necessary to call operations with side-effects for realizing axioms. Since it is not allowed to call such operations in pre- and postconditions or invariants it is impossible to express them. But test cases can easily be used to express axioms. The way from an ADT definition via an interface declaration to a test case implementation is shown in the following figures.

ADT Stack [T] -- Stack for items of type T

Types

Stack, T

Operations

New → Stack

Pop (Stack) → Stack

Top (Stack) → T

Push (Stack, T) → Stack

Empty (Stack) → Boolean

Full (Stack) → Boolean

Preconditions

Pop(Stack): not Empty(Stack)

Top(Stack): not Empty(Stack)

Push (Stack, T): not Full(Stack)

Axioms

Empty(New)

Pop(Push(s, t)) = s

Top(Push(s, t)) = t

Not Full(Pop(s))

```
interface Stack
{
  public Stack ();
    // ensure empty()
  public void pop ();
    // require !empty()
    // ensure !full()
  public Object top ();
    // require !empty()
    // ensure result != null
  public void push (Object o);
    // require o != null && !full()
    // ensure !empty()
  public boolean empty ();
  public boolean full ();
}
```

```
public class Stack_Test
{
  public void testNew ()
  {
    assert(!_stack.empty());
  }
  public void testPushPop ()
  {
    Stack oldStack = _stack.clone();
    _stack.push(new AClass());
    _stack.pop();
    assertEquals(_stack, oldStack);
  }
  public void testPushTop ()
  {
    Object obj = new AClass();
    _stack.push(obj);
    assert(_stack.top() == obj);
  }
  public void testPush ()
  {
    _stack.push(new AClass ());
    assert(!_stack.empty());
  }
  public void testPop ()
  {
    _stack.push(new AClass ());
    _stack.pop();
    assert(!_stack.full());
  }
  protected void setUp ()
  {
    _stack = new StackImpl(); // StackImpl implements Stack
  }
}
```

At first we reflected the inheritance hierarchy of the framework in the inheritance hierarchy of the test cases. Thus we could easily reuse test cases and avoid semantic shifts in sub classes. That has proved to be very useful for framework-based application development. The programmer lets his application inherit from a framework class and programs a test case which inherits from the framework test case. Then, the test case will ensure that the axioms of the framework class still hold for the application class. But the approach has its drawbacks for Java interfaces. A Java class extends exactly one class but can implement a number of interfaces. Since interfaces lead to in interface test classes we would need multiple inheritence for the test classes. Since Java only supports single inheritance, we now use delegation between test classes to reflect inheritance and interface implementation. An example is shown in Figure 7.

We don’t write negative tests for pre- and postconditions. Otherwise, we couldn’t reuse our tests for subclasses. Since subclasses are allowed to weaken preconditions and restrict postconditions, our test cases probably wouldn’t run for subclasses.

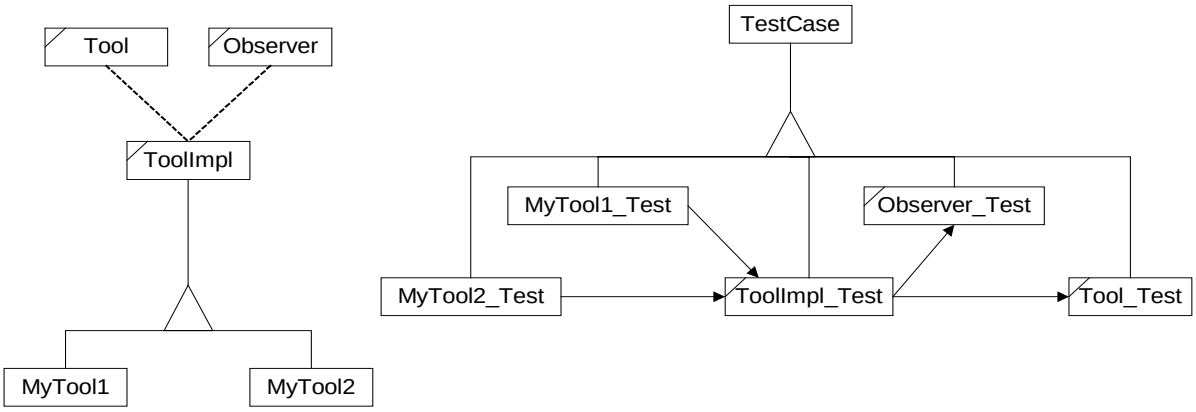

Figure 7: Use delegation between test classes to reflect inheritence between the operational class

We currently use JUNIT ([Junit99]) for testing, but we have modified it in a way, that now all test methods of a test case are run, not just the declared ones. This improvement enables the usage of template test methods inside super classes. We can define a test class for the super class inside the framework and we would like the subclass to be tested against this predefined test case which may be filled by template methods. A second improvement allows us to reflect multiple interface implementation by delegation between the test classes.

We ship the test cases together with the framework to application developers for two reasons: First, the test cases will serve as a documentation of the intended use of the framework. Second, the test cases can be used by the application developer to locate errors. During framework-based application development it is often difficult to find out whether a problem is located in the framework or in the application code. With the test cases at hand, the application developer can extend and modify them. If he detects an error in the framework he simply can send the extended test case and which we will use to correct the framework class. A general rule is: If an error is detected in the framework we first extend the test case so that the test signals the error. Then we correct the framework class.

Project Culture

We feel that the success of XP is highly dependent on project culture. We had two main proponents of XP in the framework team. They urged the team to be “extreme” all the time and not to fall back to their former behavior. This was not an easy task but we think that we now (after a year) have an XP project culture which will stay without continuos intervention.

Continuos Integration

We use an integration computer which keeps the last integrated version and the last version released of the framework. The developers have work copies of the framework on their development machines. If the framework needs to be modified, the local working version is modified. Then the modified classes are copied to the integration computer. Integration is done on the integration computer and the new version is announced to all framework developers. An integration is only valid if all test cases run through. If a new release has to be finished, the integrated version is copied to become the release version. We run the same tests for the release version like for the integration version plus some additional ones.

We make profit from the benefits explained in [Beck 99] but we also recognize that the integration process needs more and more time. The reason for this unacceptable effort is the size of the project. The framework consists of more than 850 classes and we need to compile the whole framework for the integration to ensure the right compilation state of all classes. In addition we need to test all examples of the framework. They cannot be tested automatically because of their interactive structure. But the only way to ensure the correctness of the integrated code and the whole framework after the integration is to test all examples. Our experience is that a normal integration needs approximately one hour. If you want short development cycles with continuos integration this effort of time is too much. We have no concrete solution for this problem but we try to shorten the time spend for each integration. One step towards this would be the automation of interactive tests, especially for the framework examples.

Application Migration

We are very „aggressive“ with refactoring. This results in quite large modifications of the framework from one version to the next. Applications based on the framework have to migrate to the new framework version. This may result in rather high efforts for the application development team. As the number of framework users grows, this is not acceptable. Therefore we currently work on a toolset to support the migration of application code. But in this paper we do not focus on these topics. Take a look at [Roock 2000] for a detailed discussion.

Conclusion

The XP techniques are useful for framework development. This is not an obvious result because some people argue that framework development itself stands in contrast to eXtreme programming. They argue that the very idea of frameworks contradicts the simple design guideline of eXtreme programming. We don’t think that this is true. We use the framework as a reification of our experience. It gives us the possibility to train people by using the framework. Therefore we do not insist on the design of the framework being the right one for ever and we try to keep the framework as simple as possible. We’ve described why and how we adapted XP techniques for our framework development. In using XP we have observed the following advantages:

- Higher quality of design and code of the framework.
- Less errors in the framework.
- Simpler design of the framework and therefore an improved understandability of the framework.
- Reduced size of the framework (from 400 to 100 operational classes/interfaces).

But some open issues remain:

- How can we do continuos refactoring of the framework without disturbing our customers too much.
- It proved to be a good practice to introduce new concepts or technologies in applications first, before adding them to the framework (don’t develop in stock). On the other hand the framework needs to provide leading edge concepts to maintain competitive. Therefore we need to introduce concepts which were only tested in prototypes sometimes. It is not clear what to do best in these cases.

References

[Beck99] Kent Beck. *eXtreme Programming Explained – Embrace Change*. Addison-Wesley. 1999.

[Firesmith96] Donald G. Firesmith. *Object-Oriented Regression Testing*. In: Charles F. Bowman (ed.). Wisdom of the Gurus. SIGS. New York. pp. 209-217. 1996.

[Fowler99] Martin Fowler. *Refactoring – Improving the Design of existing Code*. Addison-Wesley. Reading Massachusetts. 1999.

[Junit99] Kent Beck, Erich Gamma: *JUnit*. http://www.Xprogramming.com/

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

[Opdyke92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD Thesis. University of Illinois at Urbana-Champaign. 1992.

[Meyer97] Betrand Meyer. *Object Oriented Software Construction*. Second Edition. Prentice Hall. New Jersey. 1997.

[RiehleZüllighoven95] D. Riehle, H. Züllighoven. *A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor.* In: J.O. Coplien, D.C. Schmidt (eds.): Pattern Languages of Program Design. Reading, Massachusetts: Addison-Wesley, 1995. Chapter 2, pp. 9-42.

[Roock2000] Stefan Roock. *eXtreme frameworking – how to aim applications at evolving frameworks*. To appear in Proceedings of eXtreme Programming and Flexible Processes in Software Engineering - XP2000.

[Züllighoven98] H. Züllighoven. Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- und Materialansatz. dpunkt. 1998. German.