After the Fact

Introducing XP into an Existing C++ Project

Author: Manfred Lange, IT Consultant

Network Support Lab, Hewlett-Packard GmbH

Herrenberger Strasse 130, 71034 Boeblingen, Germany

E-Mail: Manfred_Lange@acm.org

Revision: 1.12

Last saved: April 15, 2000

Keywords

C++, Testing, Refactoring, Pair Programming, Multiple Threads

Abstract

The paper describes the journey we had to take when we introduced XP techniques into an existing project.

My intention to write this paper is to show how XP practices can be successfully integrated in existing software development efforts. In addition, I want to encourage those not already using XP, which it is fun to experiment with it, although it may take some time before you see all benefits in full extent.

I will present you my experiences with a storybook. Not only will this show you the issues that we ran into, and how we solved them. In addition, I want to show how we

started, where we started and how the mentioned XP practices could be introduced one after the other.

Of particular interest is the fact, that the system we are building consists of a set of components, which we implemented using various programming languages, among them Java and C++.

Another interesting issue was that many of the components were multi-threaded, and that one of the components encapsulates the database access, so we also have additional complexity with transactions, locking conflicts etc. Within the paper, I will concentrate on this particular database-access component, as it was the most complex part in the project and the one that caused most of the trouble in the project.

How We Started

In July, the project was in deep trouble. We were making only little progress regarding stability. For weeks we had tried to nail down the issues regarding concurrency, transactions, locking conflicts and so on. We had additional consultants from the database vendor on our team. Still, we did not see any progress.

Then I met Kent Beck and by working with me just few hours he convinced me that I would give XP a try. However, where should we start? Kent's answer to that question was "Start where you have the biggest problems."

As this was the quality, we started with test cases. Not that we had no test cases in place, but they were not sufficient as it turned out. Moreover, we had not put enough emphasis on the test cases, at least they did not play the role that XP gives them.

Lessons Learned:

1. Start where you biggest problem is.
2. If quality is your biggest issue, then start with test cases.
3. It is never too late to start using XP.

Our biggest problem was quality, so we started writing additional test cases. We used a code coverage tool in order to make sure that we did not miss any of the APIs[1] the component implemented. By only doing this, we already discovered several bugs and at the same time, we made sure that we would not re-introduce the same bugs into the code again.

Lesson Learned:

1. Some tests are better than no tests at all
2. Bad tests are better than no tests
3. Having a set of test cases gives you a systematic approach in ensuring the quality.

Testing

Having more and better test cases in place already improved our software quality. However, we had one particular issue: concurrency. In our database access component we could distinguish two types of concurrency: concurrency caused by multiple threads and concurrency caused by multiple processes accessing the same database.

Note that using XP means that you implement the test cases first. However, it our case our implementation was complete (although buggy) when we started to use test cases to that extent.

Testing Race Conditions

In addition to the unit test, that only invoke each function with a set of parameter values and checks the result against the expected behavior is not sufficient. In a multithreaded environment, as is the case when using an underlying database with transactions and locks, I also learned that I needed multithreaded tests.

With a colleague, I sat down and we tried to figure out, what the typical use case of the database would be in terms of concurrency. When would we lock an object? What kind of lock conflicts can occur? Does it make sense for a database client to wait for a lock? Are there exclusive locks? What if a thread tries to update many objects within a single transaction? This scenario requires many update locks.

Our approach to this was a set of representative test cases.

We had one test case, where we had many threads accessing the database in read-only mode, while a single thread was updating the database. All of the threads were working on the same set of data. The expected outcome was that no exception or error occurred during execution.

Another test was to have two threads running simultaneously, both reading and writing to the database. Again, we made sure that both threads were working on the same set of objects in the database. And again, the expected outcome was that no error occurred.

Yet another test included collections and queries, which required many read and update locks within a single transaction. This again increased the number of lock conflicts, resulting in a test of the error handling.

Lessons learned:

1. If adding tests to an existing C++ project, add both: the standard, well-known unit tests and concurrency tests.
2. You do not have a guarantee that the race conditions tests cover all thinkable situations. I do not know of any deterministic tests for multiple threads.

Separate Test Interface

Our database access component exposes its interfaces through a component technology[2]. So our first set of test cases consisted of functional tests, which simply did not consider the implementation.

Later we discovered that this is not sufficient. We had to add specific test cases for queries and collections. For these test cases we investigated, what set of parameters would cause the component to execute certain code in the component. We called these our white-box tests.

Finally, even this was not sufficient. It turned out, that for another category of tests, it was necessary to bypass the official interfaces. We therefor added an interface for testing purposes. This interface is available in special debug builds only. This interface was then used to access test classes for testing internal functionality, e.g. handling of transactions, distribution of objects in the database, error handling etc.

Lessons learned:

1. If you use components, have an additional test interface in place in order to bypass the official, published interface.

2. Do not only mechanically test the APIs. In addition, do consider that depending on the set of parameters passed to APIs, different code within the component may be executed.

Fully Automated Test Driver

For automatically running test cases, we use a C++ test framework that is easy to use. This test framework is comparable to JUnit. It is now in a state, that we can easily reuse it for all our other C++ projects. In addition, we can integrate it in our build process.

C++ is a language that needs to be compiled and linked. Regarding refactoring this is an issue. Interpreted languages have much shorter cycles, which provides for more cycles per time span. We addressed this problem with two means: First, we bought bigger machines with multiple processors, more memory, and faster I/O subsystems. Second, we checked our design, whether it was possible to reduce the number of dependencies between the source files. For instance, we checked all include statements, whether they were necessary. One policy here was to move include statements into the implementation files (CPP-files).

Another solution was to rethink the class hierarchies and the collaboration of classes in order to reduce the number of dependencies between source files. This also helped to make the design much easier to understand as it led to simpler interfaces of the classes.

The efforts resulted in a 50% reduction in the compile-link time.

We divided our test cases into two categories: unit tests and long-term stability tests. We executed the first set after each single change. However, as the latter took between 45 and 60 minutes to execute, we only ran them two or three times a day. The long-term stability tests also included the concurrency tests.

Lessons learned:

1. Do not underestimate the importance and usefulness of a fully automated test driver.
2. If people say, they have implemented test cases, do not trust them. Most of the time, they implemented them *after* they have completed their code. Test cases in the sense of XP have two characteristics, which distinguish them from traditional tests: First, you implement them up front. Second, they are systematic.
3. Even for compiled languages such as C++, it is possible to achieve short compile-link cycles by reducing dependencies between files and sparse use of templates, etc. This accelerates the integration of XP.

Simulating Error Conditions

A database application, especially if it is multithreaded, also raises the issue of how to handle error conditions. Among them but not limited are locking conflicts.

The database system uses an UNIX-like signaling mechanism for indicating error conditions. This means, that if a client is interested in such signals, the client has to implement a callback function also called a signal handler. Then the client registers this callback. Finally, if an error occurs, the server component (in our case the database system) invokes the signal handler. The client is free, in what the signal handler should do[3].

We wrote a class CErrorCheck that automatically detected, if the database runtime library called the signal handler during the execution of a series of database calls. If the database

runtime library called the error handler, the destructor of that class would then automatically throw an exception.

In order to test error handling, we added code to the CErrorCheck destructor, so that for debug builds, occasionally the CErrorCheck destructor threw an exception although the database system has never called the error handler.

Lessons Learned:

1. It is impossible to write a test driver for a multithreaded database component with 100% coverage. You can achieve 100% code coverage eventually. Nevertheless, you will not be able to achieve 100% coverage of all possible execution paths.
2. To improve the testing of error handling mechanisms, concentrate all error handling to a single place in your code. Then add a simulator to that place in order to increase the numbers of (simulated) errors during the execution of your test cases.

Simple Design

Some of the issues that we found while adapting XP methods caused us rethinking our design.

One thing was the implementation of collections and iterators that we needed for accessing objects in the database. At first sight, templates were the obvious solution, but unfortunately, it turned out also to cause a code-bloat. The resulting binaries were one third bigger than necessary.

Having solid test cases in place, we had enough courage to change the implementation so that we simply did not use templates any more. Within only a few days, everything was running again without problems. Without the test cases, it would have taken much longer.

Lessons Learned:

1. Avoid using templates if possible. Not using templates makes your code easier to read and understand. Use templates only, if it is your best alternative.
2. Having solid test cases makes it easier to simplify the design. The test cases tell you whether your transformation of the code was a valid one.

Another area for rethinking the design was the code for synchronizing multiple threads. Fortunately, we already had an implementation where the use of synchronization objects such as critical sections, mutexes, etc. was already concentrated in a few locations. However, we reworked it so that locks where automatically released independent on how a function is left, either through a return statement or an exception thrown.[4]

Lessons learned:

1. Simplify your design one step at a time. After each step, your perspective is different and you discover new and interesting points in your code, which deserve refactoring or a redesign.
2. Try to locate the code for a specific feature at one single place, e.g. the code for transaction handling, for distributing objects in the database, or for the management of queries.

Refactoring

Refactoring C++ is a hard task. Nevertheless, in some cases, you will choose C++ as the best alternative for implementing a component. Therefore, in this section, I will explain a few refactoring techniques for C++.

Martin Fowler describes many methods in his book [Fowler 1999][5], so I do not duplicate them here. On the other hand, I found some techniques, which Martin did not described at all, or if there were described, Martin did not focus on C++.

Replace Template Member Function By Base Class Member Function

Motivation

When implementing in C++, templates can be a powerful means for abstraction. Typical uses are collections and iterators. In some cases, however, the use of templates leads to code bloat. This is especially true, if a template class consists of many functions and/or the template takes part in nesting.

In some cases, member functions of a class template do not use the template parameter. In this case, introduce a non-parameterized base class and move the function to that base class.[6]

Mechanics

- Introduce a new non-parameterized class.
- Make the new class a base class of the template class.
- Move the member function from the template class to the new base class.
- Compile and test.

Example

Using my database access component example, I had a class that implemented queries on the database. I found it a convenient way to implement them as a template:

```
template<class T> class Query {
public:
   // other stuff left out
   int AddPredicate(string aPredicate);
private:
   std::list<string> m_predicates;
};
```

Actually, the predicate collection is the same, independent of the type for which you are actually querying. Instantiating the Query class template leads to an instantiation of 'AddPredicate()' for each type.

I introduced a new base class "QueryBase" and moved the function up the hierarchy into that class:

```
class QueryBase {
public:
   // other stuff left out
   int AddPredicate(string aPredicate);
private:
   std::list<string> m_predicates;
};


template<class T> class Query : public QueryBase {
   // other stuff left out.
};
```

With this implementation the functionality is the same, but I had only one instantiation of the function "AddPredicate()".

Replace Macros By Template Functions

Using C++ precompiler macros can lead to code bloat and bad readability. To reduce the code bloat, to increase readability and to make debugging easier, replace macros by template functions.

Motivation

The C++ precompiler provides macros. You can use macros to implement functions that you need frequently within the program, e.g. deleting an object and setting the pointer pointing to that object to NULL.

However, there are three major drawbacks for this approach: First, the use of the macro is not type safe, and second, debugging a macro can be very annoying, and third it results in code bloat.

Mechanics

- Replace the macro by a template function.

Example

Deleting an object from the heap is a frequent task in C++ programs. For additional safetyness I always check the pointer for NULL-ness, then if it is not NULL I delete the object and finally I set the pointer to NULL, so that I do not use the pointer again. The simplest approach is this:

```
if( NULL != pObject ) {
   delete pObject;
   pObject = NULL;
}
```

As this is a frequent task, my first solution to make life easier was this

```
#define CLEANUP(pointer) \
if( NULL != pointer ) { \
   delete pointer; \
   pointer = NULL; \
}
```

With this in place I could write:

```
CLEANUP(pObject);
```

This already reduced my typing work, but it still was not what I wanted. Therefor here is a better approach:

```
template<class T> void CLEANUP(void*& pointer) {
   if( NULL != pointer ) {
      delete pointer;
      pointer = NULL;
   }
}
```

This approach allows you also to set breakpoints and to step through the code. In addition, as I pass the pointer by reference I work on the original memory, so assigning NULL means, I do not assign NULL to a copy, but to the 'real' pointer.

Yet an improvement would be to make the template function inline, which eliminates the overhead for calling the function and returning the function.[7]

Pair Programming

OK, some people do not like it. Their first thought is that Pair Programming reduces productivity. No, it is not. However, there we found some issues that are interesting to note.

Social Issues

Pair Programming also means that those two persons communicate. This includes all the known issues such as misunderstanding. My opinion is that developing software is a social activity. In order to create a great product it is necessary, that the people involved know each other not only by name, but also their strengths and weaknesses, not only regarding technical skills but also regarding personal and interpersonal skills.

This helps in various ways: If you know the strengths of other people, it is easier for you to pick the best partner for the development task at hand.

One particular issue I encountered was that I face a lot of criticism regarding the underlying database. For the first time, we were using an object-oriented database. In addition, I underestimated the effort needed to integrate the product into our software. The combination of the instability and the delay lead to the criticism. Critics attributed the problems mainly to the database technology and the database product.

We had a phase, when we had all known bugs fixed and still the software did not run stable. Then – in a joint effort of the best engineers on our team - we found out, that the cause for the instability was a compiler bug.

Since then the criticism has stopped. When we joint our forces, I learned about the experience with the compiler bug, and my colleagues gained a better understanding of the database access component.

Lessons learned:

1. Working in pairs helps the first person to learn about the experiences the second person has made in other areas. The second person learns about

the software module on which the first person is working. He gets insights and eventually understands the complexity of the module.

2. Pair Programming is not only ongoing code review, but also involves social aspects. People learn about the feelings and thinking of others.

Differences in Skill Levels

Another issue might be the difference in skill level between engineers. This can lead to an engineer fearing to work with another engineer, just because the first one thinks he is not good enough. We have an engineer on our team who is an excellent C++ programmer. Another engineer had the fear that he would slow the C++ genius. You should not overlook this issue.

Lesson Learned:

Do not force people to pair program if they do not feel like it. Instead, figured out which engineers would make a good pair and take into account the fear that might be the result of different skill levels.

Summary

This paper is just a half way story. We are just at the beginning of adopting XP methods. I think, the best way is to start with just a few practices and then prove that they work. If you can prove it, you will convince the other members of your team.

However, XP is not a silver bullet and especially refactoring C++ code is not an easy task. However, with good tools, it becomes a little easier and after a while it pays off. It took us several months, but in the meantime I do not hesitate no more, rewriting code that is at the core of the database access component. I am convinced that we have good test

cases in place, which help us to reduce the impact of any changes we might want to apply.

Introducing XP practices helped us to gain the software quality we needed to finalize our product. Now a patent is pending for the product and deployment is ongoing. By mid 2000 we plan to have more than 1000 installations worldwide. I attribute this success to the introduction of the XP practices.

Acknowledgements

I would like to thank Kent Beck for spending his time at EuroPLoP 1999 and in Zurich with me. He showed me the principles of XP.

References

[Beck 2000] Kent Beck: "Extreme Programming Explained: Embrace Change", Addison Wesley 2000, ISBN 0-201-61641-6

[Fowler 1999] Martin Fowler: "Refactoring: Improving the design of existing code", Addison Wesley 1999, ISBN 0-201-48567-2

[Meyers 1992] Scott Meyers: "Effective C++: 50 Specific Ways to Improve Your Programs and Design", Addison Wesley 1992, 0-201-56364-9

Footnotes

1. API = Application Programming Interface, used here in the sense of a function with a
2. When talking of a component technology I mean either Corba or COM+.

3. The implementation of the signal handler should not execute any lengthy operations, however.
4. A pattern named “Scoped Locking” can be found at http://www.cs.wustl.edu/~schmidt/patterns/patterns.html.
5. Note especially the sidebar on page 384 “Refactoring C++ Programs”, written by Bill Opdyke.
6. This is a special case of “Pull Up Method”.
7. For further details see also Scott Meyers “Effective C++” [Meyers 1992], page 10.