Integrating Extreme Programming and Contracts

Hasko Heinecke
Daedalos Consulting AG
Seestrasse 510
8038 Zürich, Switzerland
+41 1 4810720
heinecke@daedalos.com

Christian Noack
Daedalos Consulting GmbH
Postfach 3113
58422 Witten, Germany
+49 2302 9790
noack@daedalos.com

ABSTRACT
Extreme Programming[1] (XP) is a light-weight software engineering methodology conceived by KENT BECK with a strong focus on business value. Design by Contract is a software design technique defined by BERTRAND MEYER that stresses stability and maintainability of large systems. The two are regarded as incompatible by many of their respective followers.

In this paper, the authors describe why contracts can nonetheless offer benefits to XP, and how they can be used in an XP environment. Contracts are particularly helpful in large systems development, an area that is not yet well investigated by the XP community. The authors describe how applying Design by Contract in an XP project can work, and what benefits can be expected.

Keywords
Extreme Programming, XP, Design By Contract, System Documentation, Object-Oriented Software Development

1 EXTREME PROGRAMMING

This paper assumes a working knowledge of standard Extreme Programming concepts. We will therefore not explain the various aspects of XP here. A good reference can be found in [1, 2, 3].

XP is designed to address the specific needs of software development conducted by small teams. It is less well understood how it applies to large systems development. The ideas described in this paper are an attempt integrate one well-known technique for this with XP.

2 DESIGN BY CONTRACT

A full tutorial on Design by Contract is beyond the scope of this paper. The following paragraphs give a very brief introduction to the basic concepts and the purpose of Contracts. Readers who are familiar with those concepts may want to skip them.

Pre- and Postconditions

The basic components of Design by Contract are Preconditions and Postconditions. Both are sets of logical (Boolean) expressions with no side-effects[2], that are attached to individual methods. They describe semantic properties that are required to hold when the respective method is executed.

Preconditions describe the required conditions for the method to return a reasonable result. They are checked before the actual method is executed. Postconditions describe to some extend the expected result of the method, provided the preconditions were kept. Postconditions are checked after method execution, but before returning to the calling context.

Invariants

Class Invariants are contracts attached to a *class*, that are to be checked both as Preconditions and Postconditions, and that are checked for *every* public method of a class. Therefore, most of what is said about either of the other two kinds of contracts is also valid for Class Invariants.

The remaining paper will therefore in most cases only address Preconditions and Postconditions. Class Invariants will only be discussed where they differ from the other two.

Inheritance

Contracts are, by definition (see [4, 5]), inheritance-aware: **A** subclass' methods must obey the rules given by its superclass. When class **A**'s method `m` is overwritten in subclass **B**, then the contracts for **`A.m`** must also automatically be checked for **`B.m`**.

Furthermore, subclasses must require no more than their superclass, i.e. they can only *weaken* preconditions. Conversely, they must promise at least what their superclass does. So they can only *strengthen* postconditions. This is usually implemented by combining all preconditions for a method in an inheritance chain with a logical **`or`** and all postconditions with a logical **`and`**.

[1] XP advocates usually prefer the fancier spelling "eXtreme Programming". The authors, despite their affection to orthographic idiosyncrasies, have decided to stick with the more conventional way.

[2] As a recent discussion in some newsgroups showed, the concept of side-effect freeness is somewhat hard to define. We will not discuss it in this paper but only give an intuitive definition: Pre- and Postconditions must not alter the state of the system they describe.

Implementing Contracts

A detailed discussion of different implementations is beyond the scope of this paper. Instead, we point the reader to existing products such as iContract for Java or the Eiffel language.[3]

An implementation of Design by Contract should provide some tools for formulating contracts, either in a repository or inlined in the method and class code. It should allow switching on and off contract checking for performance reasons. It should offer additional constructs beyond the standard logical operators, e.g.:

`old` in postconditions, provides access to the state of a variable at the time of the method call, i.e. when the precondition was checked. This is necessary to compare the old vs. new state of the variable.

`forall` in pre- and postconditions, checks a condition against all elements of a collection.

`exists` in pre- and postconditions, checks whether at least one element of a collection satisfies a condition.

Why Contracts?

In their article [6], JEAN-MARC JÉZÉQUEL and BERTRAND MEYER comment on the Ariane 5 disaster, where a $500 million rocket exploded about 40 seconds after take-off due to a software failure. They cite the official analysis of that incident, stating that a piece of software reused from the predecessor Ariane 4 was called in a situation that violated it's implicit preconditions, crashing the system. They claim that having the routine's contract stated *explicitly* would have made finding this violation much easier, and would probably have prevented the system crash.

While not all software defects cause so spectacular crashes, contracts do give a means to tell a developer about the constraints and promises of a piece of code. Particularly when reusing existing code that has been around for a while, or that was developed by third parties, this can give quality assurance teams a hint at what to look out for. Sometimes, contracts may even help to discover defects nobody was expecting, simply because they are enforced automatically.

Contracts vs. Assertions

The logical expressions that constitute Pre- and Postconditions are frequently referred to as assertions. For this reason, they sometimes mistaken as assertions as known from C and similar programming languages. However, they are both more and less powerful than those: They are more powerful in that they are aware of inheritance and polymorphism. They are also less powerful because they can only be attached to method invocations and returns, while assertions can be interspersed in a method's code. We have therefore mostly avoided the term "assertion" in this paper.

[3] See www.reliable-systems.com and www.eiffel.com.

Contracts can be implemented using assertions, but it takes additional effort besides writing down the plain assertions themselves. The additional operators have to be provided, and – more important – their behavior with respect to inheritance has to be simulated.

3 SIMPLICITY VS. CONTRACTS

The central coding practice of Design by Contract is the addition of contracts, expressed through preconditions, postconditions, and invariants to classes and methods. On the other hand, two of the most important coding practices in Extreme Programming are *"Do The Simplest Thing"* and *"You Ain't Gonna Need It"*. The crucial question when discussing Design by Contract from an XP perspective is:

Why would you want to add the complexity of contracts to your system?

In XP, only User Stories can justify raising the complexity of a system. So the question can be restated as: What User Stories require the addition of contracts to a system? In many software projects two such stories could be:

User Story 1: *We have several teams working on different subsystems and we want to protect their interfaces against mistakes and misinterpretations. Also, our corporate quality assurance strategy requires such protection.*

User Story 2: *We want to automatically generate some documentation on the semantics of the interfaces of each subsystem, e.g. for use by other teams and projects.*

A brief explanation why the two are reasonable and how they connect Extreme Programming and Design by Contract is given in the following paragraphs.

Subsystems in XP

Extreme Programming is usually viewed as applicable only in small teams of up to ten or twelve developers. While this is not a definitive upper limit, most people regard it as a practical rule-of-thumb for the team size. Twenty developers is regarded as too large for effectively applying XP. In his well-known work [7], FRED BROOKS says: "Adding more men [...] lengthens, not shortens, the schedule." This is taken as justification for XP's call for small teams. But in the same book, BROOKS also mentions: "This then is the problem with the small, sharp team concept: *it is too slow for really big systems.*" [7, p. 31] Quite often, systems are just too big to be developed in time by a single XP team.

One possible solution is to partition a large set of requirements in subsystems that interact only through well-understood interfaces. This may seem hard to do, but in practice a lot of business domains do have such interfaces. They are either company or industry standards, required by existing architectures, or they are imposed by third-party software such as SAP. The interface between payroll and financial accounting is an example, as are industry standards such as SWIFT for inter-bank

messaging, or separate front-office and back-office systems. Whenever there are such interfaces, they can be used to partition a system into subsystems.

In environments like that, the stories like #1 above can be satisfied by the introduction of Design by Contract. According to our understanding, contracts are most beneficial to XP projects when used for interfaces between subsystems. Through preconditions and class invariants they force calling subsystems to provide the required environment to use an interface. At the same time, the calling subsystem can rely on the interface semantics that are enforced by the callee's postconditions (and class invariants).

System Documentation

Besides run-time enforcing contracts and error notification, contracts also serve well as semantic subsystem interface documentation. They explicitly state the expected context in which an interface can do useful work. Additionally, they even specify the results that can be expected to a certain extent.

Documentation like that when done manually is notoriously incomplete and outdated. Besides, usually qualified team members are needed to produce it and their workforce is often indispensable. In contrast, contracts can be picked up by suitable tools and up-to-date documentation can be created any time it is needed, e.g. to satisfy corporate quality assurance policies. (See user story #2.)

Thus, contracts make code even more self-documenting, a goal that is clearly expressed in the XP core practice, *the source code is the design*. MEYER also mentions this aspect of contracts, e.g. in [4, p. 389]

4 XP VALUES

Besides simplicity, the three other values of Extreme Programming are also affected by Design by Contract: Communication, feedback, and courage. Communication is increased by the documentation effect of contracts, as explained above. Feedback is increased by contract checking itself, providing developers with early notice of interface misuse.

Concerning courage: Often, developers hesitate to change or refactor their own code where it is using or providing external interfaces, because they fear to break them. Contracts are a safe-guard against breaking interfaces and therefore encourage programmers to do necessary changes.

5 CONTRACTS AND UNIT TESTS

One could think that contracts are not needed when Unit Tests are written to the extent demanded by XP. However, we believe that contracts and unit tests supplement each other.

Unit tests set up a context, then perform a task and check the result. The context has to satisfy the task's preconditions for the test to work, but it never explicitly mentions those preconditions. For a person browsing through some unit tests, it is possible to derive the preconditions from the contexts that are set up. This is feasible for relatively simple test units. For subsystem interfaces, the test unit is rather complicated. In this case, implicit preconditions are rather hard to derive from the unit tests alone.

There is a partial solution to this problem: It is possible and frequent practice to write specialized unit tests that check whether interfaces fail gracefully when their implicit preconditions are *not* satisfied. However, it is hard to distinguish these unit tests from the those testing for real user stories, and it is still hard to derive the actual preconditions from them. Furthermore, multiple preconditions result in a combinatorial explosion of the number of unit tests. Therefore, unit tests cannot substitute explicitly formulated preconditions.

The relation between unit tests and postconditions is somewhat less obvious. Both unit tests and postconditions give evidence of the expected result (or resulting context) of an action. Unit tests create an example situation and check an operation's result. Postconditions, though, describe the expected result in a more general manner, albeit less detailed. As logical functions, they can describe an infinite set of possible results *and* they can even exclude impossible ones.

Furthermore, Contracts fit well into the "Once and Only Once" principle of XP. They represent the *universal* (desired) semantics of a method, whereas Unit Tests only represent its semantics under a given assumption. Therefore, associating the contracts with their methods instead of their multitude of tests is following that principle.

As has been illustrated, Unit Tests and Contracts address different, if somewhat overlapping issues. Unit Tests are a core practice of Extreme Programming. Their presence is required throughout the life-cycle of every XP project. Contracts are not crucial for XP to work. However, in a large system development effort that is partitioned into several loosely coupled subsystems, they can help ease communication difficulties that arise at their interfaces.

6 CONTRACTS AS AN IMPLEMENTATION PATTERN

The inventor of Design by Contract himself, BERTRAND MEYER notes that the contracts look formal and "shocking to most" [4] who first encounter the concept. Of course, he adds that they are the foundation of code stability and other quality goals. The authors agree in that they are indeed a means to achieve those goals, where it would be hard with Unit Tests alone.

However, contracts are not as strange to developers as they might appear at first glance. In fact, they are an implementation pattern commonly found in existing code.

Assertions Again

Contracts are a particular form of assertions, as explained

earlier. Assertions themselves are not unknown to most developers, and they are frequently used by the more careful in large systems projects. Plain assertions are sometimes considered unnecessary in the presence of Unit Test suites, but they are hardly a strange and new concept.

Preconditions

Preconditions are found in many methods in their disguise as Guard Clauses[4]. Using Preconditions instead makes them easier to locate and additionally enforces sensible rules for methods overwritten by subclasses.

Postconditions

Postconditions are often simulated by methods checking a message's result. This is commonly used when the message sent belongs to another module or is supposed to call a different application.

Code like this expresses the (healthy) mistrust in the developers who wrote the called code. However, it would rather be the other code's responsibility to enforce the promises made, if only to have it in one place instead of dispersing it through the client code. This only works, when those promises are documented, which postconditions do better than API documentation.

Class Invariants

Class Invariants are not very frequently encountered in existing code. However, they are sometimes present without being explicitly formulated. Then, they are an annoying source of what is often called "beginners' faults". An example for this is the **`equals`**/**`hashCode`** relationship in Java[5]:

When you overwrite **`equals()`** in a class, you have to make sure that **`equal`** objects still return the same hash code. However, this is enforced nowhere, and it's only stated in the class library documentation and therefore easily missed by beginners.[6] Of course, the veteran developer will always remember it, unless they are in a hurry.

Again, having explicitly stated Invariants will help locate and enforce them. Developers *encountering* class invariants (as opposed to those writing them) will probably not find them strange and burdensome but rather welcome them as an additional aid.

[4] A Guard Clause, as described in [8] is a condition that is checked at the beginning of a method, and that raises some kind of exception when it fails.

[5] The same relationship exists in Smalltalk.

[6] Interestingly, this is considered a class invariant of class **`Object`** by both the Java and Smalltalk documentation. Actually, it should rather be an invariant of an interface that specifies hashing. However, there is no such interface in Java and of course none in Smalltalk.

7 CONCLUSION

Design by Contract is certainly not a core practice of XP, but neither does it contradict the XP values. It has been shown that Design by Contract does offer benefits in an XP environment, if wisely applied. In particular, large systems efforts can use Contracts to specify and document their subsystems' semantics to a certain degree.

The authors are currently applying the approach described in their projects. An empirical analysis of its success will show to what extent the described effects will really benefit large XP projects.

REFERENCES

1. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley Longman, 1999
2. R. Jeffries and Ann Anderson, *Extreme Programming Installed*, Addison Wesley Longman 2001
3. K. Beck and M. Fowler, *Planning Extreme Programming*, Addison Wesley Longman, 2001
4. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 2nd ed., 1997
5. B. Liskov, “Data abstraction and hierarchy”, SIGPLAN Notices, vol. 23, May 1998
6. J.-M. Jézéquel and B. Meyer, “Design by contract: The lessons of Ariane”, *Computer*, vol. 30, pp. 129-130, January 1999