Agile Software Development of Embedded Systems

Version : 1.0
Date : 2007.02.12

Authors
Andrew Wils
Stefan Van Baelen

Status

Final

Confidentiality

Public

Agile Deliverable D.2.14

Software Architecture and eXtreme Programming

Abstract

Extreme Programming is arguably the most mature but also the most prescriptive agile methodology, using almost religiously followed mantra's and practices to guide software development. This document describes how XP treats architecture, and how architecture can be developed in XP. We will pinpoint possible entrypoints in the XP process and explicit possible XP arguments or dangers against architecture.

ITEA

INFORMATION TECHNOLOGY

FOR EUROPEAN ADVANCEMENT

Software Architecture and eXtreme Programming

Andrew Wils and Stefan Van Baelen
K.U.Leuven
Department of Computer Science
Celestijnenlaan 200A, 3001 Leuven, Belgium
{ andrew | stefanv } @ cs.kuleuven.be

February 12, 2007

Abstract

Extreme Programming is arguably the most mature but also the most prescriptive agile methodology, using almost religiously followed mantra's and practices to guide software development. This document describes how XP treats architecture, and how architecture can be developed in XP. We will pinpoint possible entrypoints in the XP process and explicit possible XP arguments or dangers against architecture.

The XP process has a unique set of values, principles and practices (such as customer collaboration and team communication) that interact with each other. Together they have proven to bring some interesting benefits to software development. By planning with the customer and working with small iterations, software project slips are much reduced. Also, the many XP code-centric development principles tend to deliver robust and easy-to-extend software.

For a more elaborate explanation why this is, we refer to the many books that have been written on XP, e.g. [1], [2], [3], [4], [5], [6], [7].

This document covers three aspects of XP and architecture. First we show architecture as it is in XP, meaning how architecture is treated and addressed in XP. Next, we identify hooks in the XP process where we can add additional support for architecture and its documentation. Finally, we discuss the dangers of inserting architecture-centric methods in XP. We show where these methods might break down the XP process, and where they are hard or perhaps impossible to apply.

1 Architecture as it is in XP

A first, perhaps naive look at XP reveals that all architecture and software design is done in something that is called a *quick design session*. However, these sessions only deal with coming up with a simple design for a particular sub-problem. They typically last 10 tot 30 minutes and discard all produced artefacts. Moreover, quick design sessions are not even mandatory: Ron Jeffries enigmatically suggests to "prefer to let the code tell what it wants to be" [1] . This emphasis on the code is typical for XP. The code is the software system and therefore the most important artefact, known and understood by everyone

on the software team. It is also at the code level that we need to look for architecture. If everyone knows all of the code, they implicitly know the architecture. This, XP argues, eliminates the need for persistent architectural and design diagrams, and all other forms of documentation. As for the construction of an initial architecture, Wake [5] argues that XP places less emphasis on up-front architecture than other methods because architecture has less "impact" on the software process. XP claims to overcome hard-to-change architectures because XP embraces change. On the other hand, Fowler writes in his essay "Is design dead?" [2] that there is room for design before coding. As an example, you can use UML diagrams in XP, but most often not as an artefact. Models are deemed unreliable, because they tend to be forgotten or tend to get inconsistent after a while.

XP does offer a few practices to help communicate and construct an architecture though. Architecture shows up mainly in spikes, the metaphor and the first iteration. Spikes are used to test out a preliminary architecture. Instead of validating an architecture for the right non-functional patterns, quality attributes are quickly tested out in a quick throw-away solution (e.g a simulation). In this way, spikes serve as a first deployment (diagram). The metaphor is an XP practice that is supposed to provide a common base of understanding about the key concepts in the system and their interactions. In this way it acts as a simplified architecture. However, metaphor is one of the least applied XP practices because it is not easy to come up with a suitable metaphor. First Iteration is the implementation of a first set of user stories, Kent Beck explains [6], that are chosen to force create the whole system. It is supposed to be a skinny, installable and configured version of the software. First iteration is also expected to deliver the initial architecture of the system.

Having an initial architecture is not sufficient, especially using a process that embraces changes so much as XP does. In XP, the small releases practice should detect and fix architectural errors quickly, because of early feedback in real use. Also, XP encourages larger refactorings: changes in the software that do not affect functionality. This follows from the XP observation that you cannot do everything right from the beginning and the principle that coders should be designers as well (and vice versa). The latter principle ensures that programmers won't try to hack their way around the design and that designers are always in touch with the code. Refactoring avoids introducing new bugs by relying on continuous and automated testing and integration.

What about quality attributes in XP? XP believes it covers these in its user stories. These stories can be compared to use cases. Crispin [3] suggests to ascertain the required quality for each story by asking oneself questions such as "what is the worst thing that can happen in this story?" As each story gets implemented one at a time, the associated quality attributes are dealt with incrementally as well. XP's frequent releases ensure that the developers and customers can validate these attributes and solve problems early in the development process.

We will end this section with an example of how detailed an architecture is worked out in XP. Newkirk [7] describes a typical architecture that is "sufficient" to be able to generate estimates of the stories. It consists of a drawing containing 6 icons (amongst them a web client, server, servlet and database) accompanied by a 6 line-explanation.

2 XP dangers and opportunities

Nord [8] describes the advantages of architecture-centric methods such as attribute driven design (ADD), ATAM and CBAM), when applied to XP. However, care must be taken when applying these methods. When compared to RUP, the architectural baseline in the sense of component diagrams is shared orally in XP. This has a number of dangers. A review of the architecture will be more difficult, because the latter is implicit. Therefore either more team communication will be necessary for this, or more architecture documentation. Likewise, some dangers lie in handling quality attributes in user stories. Functional behavior is maintained throughout the development with the use of automated tests. In particular, continuous integration and refactoring rely heavily on this automated testing to ensure that the old functionality still keeps working. This requirements validation is often not possible with quality attributes, yet quality attributes must be maintained as the software changes. If there are no tests in place for the quality attributes of the system, they cannot be guaranteed, especially as the architecture is constantly changing with the code. Hence, the relevant stories cannot be discarded once they are implemented.

Some XP proponents are also concerned with some issues that might come from embracing change. For example, Wake [5] is a bit anxious about two issues where this principle might show its weaknesses: changes in threading and synchronization and changes in the data model. These issues can be partly blamed on not working out the architecture.

These dangers suggest that we make architecture more explicit, or change make the architectural methods more agile, or both. Making architecture more explicit can guide the development of larger projects. The maximum size of XP teams is around 12 people. Larger sized project need more or larger teams, and oral communication won't suffice anymore. In this case, more architecture-centric methods have to be used to coordinate the teams. For example, the subsystems and/or components could be divided amongst a number of XP teams. Then, each required interface would involve one XP team playing the role of the customer. It can also help to address quality issues more thoroughly.

Conversely, Eckstein and Katzenberg compared XP and RUP in [2], stating they are convinced that the two can be combined. With this they mean that it is possible to apply the XP process and practices inside the RUP phases and still respect RUP. We could do more and only apply relevant RUP practices when we need to. This would help to avoid the pitfalls of using "heavy" processes and the risk that increasing the team size only further postpones the project completion date [9]. also, or consequence?: It would even be more beneficial if architecture could be made more XP-like. That is, if architecture can be treated and tested in the same way as code. One step in this direction is "extreme modeling". Boger et al describe in [2] how models can be executed and tested. Quality attributes too must be included and tested in these models. Code should also be more strongly linked with the architecture. Consistency could e.g. be ensured by navigation from the code to the architecture or automatic "reverse engineering" the architecture.

3 Conclusion

It is clear that the XP process has certain benefits for small and medium sized software development projects. On the other hand, no large system can be constructed without an architecture that has been made explicit. Quality attributes are useful for projects of all sizes. The weakness of XP is perhaps that it does not forbid architecture, but that it does not encourage architecture. Unfortunately, the advantages of XP are difficult to transfer to architecture. XP relies on the interplay of an number of practices and principles that are easy to implement for code based development and difficult to apply on anything else. Shifting the balance of design and coding can easily disturb the efficiency of XP. Nevertheless, we showed a number of ways to add more architecture-centric design to XP.

References

[1] R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*. Addison Wesley, 2001.

[2] G. Succi and M. Marchesi, *Extreme Programming Examined*. Addison Wesley, 2001.

[3] L. Crispin and T. House, *Testing Extreme Programming*. Addison Wesley, 2003.

[4] K. Auer and R. Miller, *Extreme Programming Applied*. Addison Wesley, 2002.

[5] W. C. Wake, *Extreme Programming Explored*. Addison Wesley, 2002.

[6] K. Beck and M. Fowler, *Planning Extreme Programming*. Addison Welsey, 2001.

[7] J. Newkirk and R. C. Martin, *Extreme Programming in Practice*. Addison Welsey, 2001.

[8] R. L. Nord, J. E. Tomayko, and R. Wojcik, "Integrating software-architecture-centric methods into extreme programming," tech. rep., CMU/-SEI, 2004.

[9] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1995.