Synopsis

Architecture and Design in eXtreme Programming: *The Architectural Game,* introducing a new practice

This work is an article with the following abstract:

> This article presents a new practice to the agile method eXtreme Programming (XP): *The Architectural Game*. The goal of this new practice is to incorporate architectural planning to XP thus ensuring an adequate sustainable architecture. Traditional development methods seek to establish an architecture that is expandable and maintainable in the future through careful planning whereas XP only implements an architecture sufficient enough to support the current snapshot of the system. The new practice uses the user stories at hand in the current iteration to implement an adequate sustainable architecture.
>
> From an Action Research (AR) based on a case study of a software development project using XP, we argue that XP does not necessarily guarantee an adequate sustainable design. The criteria used to evaluate the design are found from traditional development methods, and compared with design criteria from XP, assessing that they are compatible. This leads us to propose the new practice. An investigation of whether or not the new practice conforms to the values and principles of XP, shows us that The Architectural Game can be incorporated into XP without breaking fundamentally with the values.

20th June 2005, Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark

Authors

Rolf Njor Jensen

Thomas Møller

Peter Sönder

Architecture and Design in eXtreme Programming: "The Architectural Game", introducing a new practice

Rolf Njor Jensen
rolf@cs.aau.dk

Peter Sönder
sunlock@cs.aau.dk

Thomas Møller
molz@cs.aau.dk

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark

20th June 2005

Abstract

This article presents a new practice to the agile method eXtreme Programming (XP): The Architectural Game. *The goal of this new practice is to incorporate architectural planning to XP thus ensuring an adequate sustainable architecture. Traditional development methods seek to establish an architecture that is expandable and maintainable in the future through careful planning whereas XP only implements an architecture sufficient enough to support the current snapshot of the system. The new practice uses the user stories at hand in the current iteration to implement an adequate sustainable architecture.*

From an Action Research (AR) based on a case study of a software development project using XP, we argue that XP does not necessarily guarantee an adequate sustainable design. The criteria used to evaluate the design are found from traditional development methods, and compared with design criteria from XP, assessing that they are compatible. This leads us to propose the new practice. An investigation of whether or not the new practice conforms to the values and principles of XP, shows us that The Architectural Game can be incorporated into XP without breaking fundamentally with the values.

1 Introduction

Since the late 90s there has been a huge interest in the field of lightweighted methods. These methods are best known as agile methods, where XP has attracted the most attention. Agile software development emphasizes on close collaboration between the developer team and the customer through

face-to-face communication, frequent delivery, self-organizing teams, and rapid response to changes in requirements (Beck, 2000).

Non-agile software development methods such as the Unified Process (UP) (Jacobson *et al.*, 1999) produce substantial documentation during the development, and the architecture, design, implementation, and test are based upon a fixed set of requirements determined early on in the process. In contrast, XP and other agile methods produce no other documentation than the code itself, and due to core practices such as weekly cycle, incremental design, test-first programming, and continuous integration they are able to respond without significant overhead to changes in requirements.

Does XP produce a sustainable architecture? This question has frequently been testet. Agile methods are creating their design by constantly redesigning through refactoring (Beck, 1999, 2000, 2004; Fowler, 2004). This way the architecture will keep on improving throughout the development phase, but only on demand. Agile software developers will not produce any part of the system, which the customer has not yet requested, because they might not request it after all. This is also known as the YAGNI principle or "You Aren't Going to Need It" (Fowler, 2004).

However, a case study using XP showed us that the produced architecture in XP is not necessarily an adequate sustainable architecture. Furthermore, some work (Fowler, 2004) suggests that the approach to design in XP might be too restrictive. This leads us to suggest the introduction of a new XP practice - *The Architectural Game*. We suggest a more systematic approach to laying out the architecture.

Is The Architectural Game violating XP as a method? In order to answer the question we will compare the values and principles of XP with The Architectural Game. By carefully comparing each value and principle with The Architectural Game, we can determine that this new practice does not conflict fundamentally with XP.

1.1 Contribution of this Article

- A case study shows that using XP does not necessarily produce an adequate sustainable architecture.
- An addition to XP's portfolio of practices: The Architectural Game.

1.2 Overview

The rest of this paper is organized as follows: In Section 2 we present an overview of the field of research related to XP, especially related to design and architecture. Section 3 presents a case study using the XP method. In Section 4 we argue that traditional design criteria apply to software developed using XP. We use these criteria to show that the design from our case study has substantial flaws. In Section 5 we propose the new XP practice, *The Architectural Game*. Section 6 investigates the applicability of The Architectural Game with respect to the values and principles of XP. Section 7 discusses the validity of our obtained results. Finally, Section 8 outlines

work still needing to be done regarding testing and evaluation of the new practice.

2 Agile Software Development

The "Agile Software Development Manifesto" (Beck *et al.*, 2001) is a manifest signed by the originators of a large number of these methods which state the values that are shared by all agile methods. XP is one of these agile methods. It was first presented in an article (Beck, 1999), and shortly after a book was published introducing XP, and for the following years serving as a reference for XP (Beck, 2000). Since then XP has been subject to much interesting work, some of which is introduced unorganized in the following:

In a master thesis (Kalermo and Rissanen, 2002) the applicability of the Agile Manifesto is analyzed through a case study and a literary review. They have used XP and an in-house agile method, and concludes that *"...traditional and agile methods can also be combined to gain the best results."*

The practices of XP raises some serious concerns regarding their role in conceptual modeling and code generation (Juric, 2000), which directly affects software architecture solutions. Furthermore, Juric addresses similarities between Rational Unified Process and XP.

The lack of explicit requirements in XP produces several problems, and some work has gone into researching how a requirement model can be incorporated into the XP method (Leonardi and do Prado Leite, 2002; do Prado Leite, 2001).

The 12 practices (Beck, 2000) in XP supports the underlying fundamental values. The relationship between values and practices has been subject to some investigation (Robinson and Sharp, 2003), suggesting that other practices might just as well support the same values.

Using XP in a student setting provides some challenges. Based on a case study, it is suggested (Smrtic and Grinstein, 2004) that pair programming, unit testing, and just-in-time design are beneficial.

The approach to design in XP is radically different than in previous methods. The role of design in XP has been elaborated and compared to traditional ways of design (Fowler, 2004). Also, new challenges emerge when continuous design is combined with the traditional "Big Design Up Front" (BDUF). Specific design goals and rules of thumb apply in an XP setting (Shore, 2004).

Beck has recently revised his description of XP (Beck, 2004), giving - among other issues - new attention to design, pointing to *Simplicity* as the fundamental principle, and *Appropriate for the intended audience*, *Communicative*, *Factored* and *Minimal* as criteria to evaluate design. Beck (2004) also states that the XP strategy is *Enough Design Up Front* (ENUF), and continuous incremental design. Moreover, Beck (2004) states that XP should be adapted to the environment - adding and modifying new practices

which are needed to support the underlying values and principles.

3 A Case Study

We have performed a case study of a software development project using XP. In collaboration with another research group, we have performed the development ourselves. The development team consisted of five developers, and the product was a search engine able to perform searches across several SOAP sources.

The project scope allowed for one XP iteration of: a round of *the planning game*, *pair programming*, *refactoring*, *small release*, *test-first programming*, *collective ownership*, and *customer involvement*. We followed XP as described by Steinberg and Palmer (2004), which follows the original description of XP (Beck, 1999).

3.1 Action Research

We have used Action Research (AR) as the approach to reason about the case study. AR has proved useful in researching real-world phenomenon (McKay and Marshall, 2001). In contrast to other research methods like e.g. laboratory experiments, AR takes place in the real world. This hands-on approach ensures a close coupling to the practical problem thus ensuring relevance in collected empirical evidence.

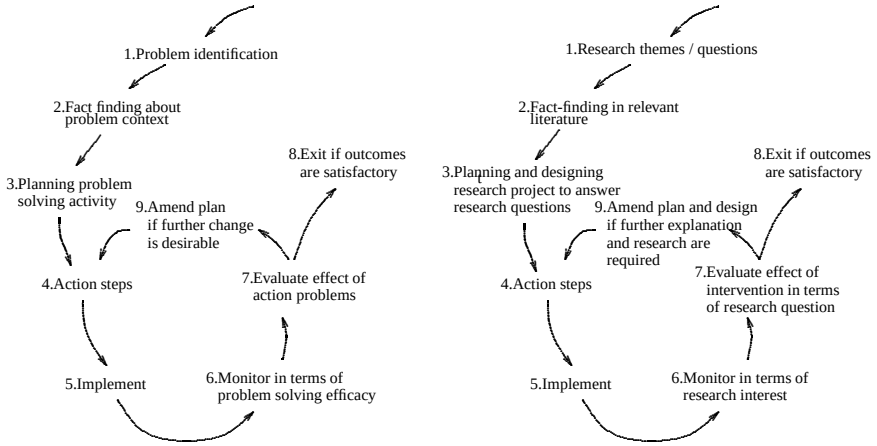
AR is not without its critics. There are several arguments why AR may be disfavored in academic circles, where research is evaluated according to traditional scientific criteria - meaning that they are repeatable with the same results. However, the strengths of AR compared to other research paradigms is more significant than its weaknesses (McKay and Marshall, 2001).

AR is in its name self-explanatory. AR provides a dual aim by being both a method for solving practical problems; action, and a method for generating and testing theory; research. This means that the action researcher has dual aims. The researcher must seek to bring improvements to a problem and simultaneously generate new knowledge and new insights through the performed activities (McKay and Marshall, 2001). The two cycles are illustrated on Figure 1. These cycles are performed in parallel, thus one overlaid with the other.

3.2 How we used Action Research

This section will explain how we used AR in our project. First focus will be on solving the problem “action” and hence on the “research”. The emphasized number in parentheses refers to cycle labels in Figure 1.

Action: Prior to the start of the XP development project, we became aware of the development problem (*1*), and during a preliminary meeting with the client (stake holders), problem context and scope (*2*) was discussed. Hereafter, the XP project was started (*3*) with a round of the plan-



(a) Action Cycle (b) Research Cycle

Figure 1: The dual cycles of Action Research (McKay and Marshall, 2001)

ning game, and the first iteration was kicked off (*4*, *5*). During the XP iteration, we used automated unit tests (*6*). The XP iteration ended with delivery to the customer (*7*). At this point, the scope of the project in terms of time did not allow for another XP iteration, and thus the action cycle was exited.

Research: Several aspects of XP was in focus prior to the project (*1*), including quality of application design. Review of XP-related literature (*2*). Before the start of the development project, we set up frameworks to collect empirical evidence using diaries and source code produced during development (*3*). Hereafter we wrote diary entries (*4*, *5*), and continuously re-read diary entries (*6*). The cycle ended with an evaluation of the data collected, which showed that there were problems with the architecture (*7*).

3.3 Collecting Empirical Evidence

We collected empirical evidence in two ways. First we wrote entries in a diary on a daily basis based on a template. Furthermore, we had the main product of the project, being the source code.

Diary: From Jepsen *et al.* (1997) we have a list of observations on the use of diaries in a development project. The article argues that the use of diaries encourages the establishment of new working habits and enhances the participants understanding of the methods applied. Furthermore, a diary is good for clarifying problem solving situations and useful when evaluating projects and project situations. Diaries support the interplay between planning and evaluating activities.

Jepsen *et al.* (1997) has a list with practical advice. Key values in writing diaries include being consistent in writing and ensuring a well formed layout. As a result of this, we chose to write each day at noon and then re-

fine it just before the work-day ended. To ensure the quality of the entries, we made a template. This template held the questions we asked during our initial research phase.

Code: The main product of the action part of our research was the actual program. A Concurrent Version System (CVS) was used to manage the source code. This enabled us to extract daily revisions of the code.

3.4 Research Question

Reviewing our empirical evidence, it was clear that the design produced in the development project had flaws. Our research problem was therefore to determine whether using XP always ensures a sustainable architecture, and if not, which measures could be taken to ensure it.

4 Evaluating Design and Architecture

Before we can determine whether or not an architecture is adequate sustainable, we need to specify what this means. For this purpose we will be using the work on quality metrics initially from the 70s (Gilb, 1988). This work has been cited throughout textbooks and articles, and most of these metrics (or criteria) are still used in object oriented software development.

There are many aspects of how to achieve a sustainable architecture through careful designing (Gilb, 1988). We will only be focusing on the design criteria mentioned in a textbook on Object Oriented Analysis and Design (Mathiassen *et al.*, 2001). After all, when Gilb first published his software metrics, object oriented software development still laid well in the future. Therefore some of the metrics he put forth have their roots in imperative programming, and they are of no use to us.

A good design should not only be judged from these criteria, but also on the absence of weaknesses. Therefore, when evaluating your design, it is important to remember that a good design has no essential weak points. This quality principle was first formulated in 1964 (Alexander, 1994), in a work on physical architecture. It has become a frequently used quality criterion in object oriented software development (Mathiassen *et al.*, 2001).

4.1 Quality Criteria for Architecture

Among the criteria presented by Mathiassen *et al.* (2001), *usability*, *flexibility*, and *comprehensible* are determined as being generally applicable to most software system.

Usability, whether it is an application with an user interface or as an application program interface (API), it should be easy to use. **Flexibility**, because a system should be able to be changed once implemented. When a project is close to being finalized and the customers requirements to the system suddenly change, it should be possible to adapt these new requirements without any substantial cost in time or price. **Comprehensible**, that a minimal effort is required to obtain a coherent view of the software.

Moreover, the concepts *coupling* and *cohesion* are criteria that apply to any software and among some of the earliest used (Kalermo and Rissanen, 2002; Fowler, 2001).

In XP simple solutions are preferred when possible. Four criteria are given to evaluate the simplicity of a design (Beck, 2004): *Appropriate for the intended audience, Communicative, Factored* and *Minimal*.

However, we believe that the criteria presented earlier originating from traditional methods still apply in an XP setting:

- *Usability*: One of the core principles of XP is economy: That the software provides value to the customer. If the software is not usable, it provides no value for the customer.
- *Flexibility*: XP promotes embracing change, and is built upon the understanding of changing requirements and the necessity of the methods ability to adapt the software to changed requirements. A flexible system is indeed changeable without any substantial increase in cost.
- *Comprehensible*: Comprehensible maps to "Appropriate for the intended audience" (Beck, 2004):

  > "*It doesn't matter how brilliant and elegant a design is; if the people who need to work with it don't understand it, it isn't simple for them*"

  And "Communicative":

  > "*... Like words in a vocabulary, the elements of the system communicate to future readers.*"

  If the system is not comprehensible, it can not fulfill "Appropriate for the intended audience and communicative".
- *Coupling and cohesion*: A design that is loosely coupled and has high cohesion achieves the same properties that are implied by "factored" and "minimal".

4.2 Review of Design

We have reviewed the design of the source code in the case study with respect to the quality criteria above, and have found several issues where the design does not meet the criteria:

Usability: The product presented two interfaces to the user - a web interface and a SOAP interface. The user interfaces (UI) were designed as the simplest that could possibly work. Each of the screens were able to handle the prescribed function, but no thoughts were made on how to map them together. This decreased the usability of the system significantly, since there was no starting point for the application, and no way to get from one screen to another in the user interface.

Flexibility: This could have been reached through the use of patterns. Revising our code, we found that we did not implement any design patterns. In general, layering is not taken into consideration in the design of our system. A Facade-pattern could be useful to hide the database implementation ensuring both flexibility and portability. Database connections could be handled with a Singleton-pattern thus eliminating the need for the implemented decentralized calls to open- and close database connection.

Coupling and Cohesion: The layering of the application is not as clear as it could have been. The interaction with the database is done with database-specific details hard coded into the model classes. This ties the persistent storage layer closely to the model classes - a case of high coupling.

4.3 Small-scale Refactorings

During the course of the case study, continuous refactoring was done in small-scale. This is also reflected in our diary. One entry notes that:

> *At the end of the day (approximately 16 o'clock) Jacob and Rolf started implementing task 8.8, which triggered some refactoring of adjacent code (Postgres classes, etc).*

Although these refactorings were performed the architecture still had the flaws mentioned in Section 4.1.

4.4 A Flawed Architecture

In summary the case study produced software which has shortcomings with respect to the quality criteria mentioned above. Giving that the practices of XP where followed, this leads us to believe that ***XP does not always produce an adequate sustainable architecture***.

5 The Architectural Game

Our analysis suggests that XP could be augmented with a practice to support a more institutionalized approach to laying out design and architecture of the application.

We propose a new practice to XP, *The Architectural Game*. The aim of The Architectural Game is to bring the architecture and design of the application in focus.

The traditional design method is doing BDUF, but Beck (2004) specifies that XP is not "no design", but rather ENUF. However, the governing principle of design is still:

> "*The simplest solution that can possibly make it work.*".

XP does not incorporate more design than strictly needed to get the current story working. On the contrary, Fowler (2004) argues that:

> *"... there is a role for a broad starting point architecture. Such things as stating early on how to layer the application, how you'll interact with the database (if you need one)...".*

This is in accordance with the proposed practice. The proposed practice does exactly that - provides the development team with an opportunity to sketch general directions of the design, laying out patterns, layering of the application, etc.

During the planning game and the continuous communication the developers become aware of the functional as well as the non-functional requirements to the application. These requirements are the base of the knowledge upon which the developers should create the architecture.

Typically functional requirements are expressed in user stories (Beck, 2004). Non-functional requirements - such as performance requirements, portability requirements, etc is more likely to be expressed in acceptance tests. Some requirements, non-functional and functional alike, might not be expressed at all in the artifacts produced during the course of an XP project (Davies, 2001).

Traditional software development has more focus on the non-functional requirements. Larman (2004) argues that non-functional requirements are structural elements needed for a sustainable architecture.

Requirements to the software are in XP expressed through user stories, acceptance tests and communication with the customer. During this new practice, The Architectural Game, developers should - in face of the known requirements in general terms lay out an architecture for the upcoming iteration.

5.1 When to Play?

The Architectural Game should be played at the start of the first iteration. During the course of development several occasions may arise that call for a new round of architectural gaming.

It is notable that The Architectural Game should not be played in every project. It should be played whenever the design criteria for the system requires a more longterm planning of the architecture. For instance, using *traceability* might require more architectural planning than *comprehensibility*.

The chapter "Whole Team" (Beck, 2004) describes the role of the architect on the XP team. The architect is part of the team and works on stories regarding equal terms with the other developers. However, he is always seeking a need for large-scale refactoring. When the architect - or some other member of the team - notes the need for a large-scale refactoring of the application, it is time to play The Architectural Game.

At some point in the course of the project, a subset of the system might be finished. A new part of the system is up for implementation. This will be a good time to play the game. The team is entering unknown territory, and it is an opportunity to consider which challenges might lie buried.

5.2 How to Play?

The key criteria for design in XP is simplicity. This should also be the governing criteria in The Architectural Game. However, The Architectural Game provides time to consider the classical criteria.

Process: The general idea of how to play is: Get an overview of the known requirements. Determine which criteria are important to satisfy. Using these requirements and criteria, come up with a suggestion to an architecture. If this is not the initial round of the game, consider which problems that the current design embodies. Review the proposed architecture, and if it isn't adequate sustainable, the team tries to get an overview once more, coming up with a new suggestion to an architecture and reviewing it.

Accepted Responsibility: As every other activity in XP it is important to ensure that people actually accept the responsibility given to them. Collective code ownership, of course, extends to the design and architecture which means that The Architectural Game is an activity in which all the developers participate. Making The Architectural Game a joint practice, it is also possible to draw on the cumulative experience of the team. Instead of having the design decision made by a pair of programmers who happen not to have any experience with this particular problem, someone might have experience with the particular problem and be able to contribute with this experience.

Communication and Tools: It is important that communication between the players of the game is clear and conveys intentions about design. In UP (Jacobson *et al.*, 1999) and OOAD (Mathiassen *et al.*, 2001) a number of tools are used to express architectural ideas and explore solutions to design issues. Some of these may be adopted to the Architectural Gamebut with some considerations in mind: First of all, which tools are applicable depends on the domain of the application. Second of all, most of the tools produce some sort of artifacts. Beck (2004) has a strong view about documentation - it should be kept to an absolute minimum. Minimal documentation is also an issue in one of the values mentioned in the Agile Manifesto (Beck *et al.*, 2001). Fowler (2004) mentions diagrams and recommends that you can use Unified Modeling Language (UML) and Class Responsibility Cards (CRC) - if you keep in mind that communication is the reason for drawing the diagrams in the first place. They should only contain important elements and not be exhaustive. Furthermore, whatever artifacts are produced they should not be preserved but thrown away when not used anymore. Considering the practices of XP as described by Beck (2004), playing The Architectural Game can then act as enabling practice for the practice *Informative workspace*, since the artifacts produced during the game can (and should be) put on display so everybody can see them, and alter them as they see fit.

6 Augmenting XP

Augmenting XP with new or altered practices is supported by Beck (2004), provided that the new or altered practices support the underlying values and principles of XP. The values of XP are the ones that the development team have to embrace to actually do XP. However, the values are not close enough to the daily development to provide guidance in everyday problems. Between the values and the practices Beck describes the principles that connect them. In the following we will go through the values and principles of XP, considering how The Architectural Game fits.

6.1 Values

These are the values of XP as Beck (2004) describes them.

- *Communication:* The Architectural Game strengthens communication. As such, it is an enabling practice because it encourages communication between the team members regarding the design and architecture.
- *Simplicity:* Playing The Architectural Game encourages simplicity. It is the governing design principle. Whenever the design of the application grows complicated, e.g. with high coupling and/or low cohesion, it is an opportunity to stop and play the game.
- *Feedback:* The feedback value evolves around one of the key features of XP, *Embracing change*. Embracing change leads to the need for feedback: Was the change right? The Architectural Game tries to forecast the direction of development a little further than the current tasks, but the practices enabling feedback are still in play: Test-first programming, continuous integration, stories, and real customer involvement.
- *Courage:* The Architectural Game benefits from courage. Because the team members have the courage to act upon problems with the architecture. They are willing to play the game, and redesign the application. Courage is the investment the team must take when playing the game.
- *Respect:* Equality of the team members and respect for the project is a value that lies below the surface of the previous values. If The Architectural Game is played by everybody, with everybody participating, it does not violate this value - it might even support it.

6.2 Principles

In the following we will consider an extract of the principles which we consider relevant to The Architectural Game. These are split in two: those who are not conflicting with the principles and those who are conflicting.

Non-conflicting:

- *Economics:* In the XP context, the Economic principle states that it is business needs that should drive the development (Beck, 2004).

  > *"Software development is more valuable when it earns money sooner and spends money later. Incremental design explicitly defers design investment until the last responsible moment in an effort to spend money later."*

  The Architectural Game could seem to violate this principle by forecasting the future and making general design decisions on beforehand. However, The Architectural Game only tries to forecast based on knowledge gained from communication with the customer and from user stories. The Architectural Game is therefore not about specifying the design in exhaustive detail, and imagining a comprehensive list of requirements that the design should satisfy, but rather utilizing the known requirements in developing an adequate sustainable architecture.

- *Mutual Benefit:* Does The Architectural Game provide mutual benefit? - For developers it is an opportunity to sketch a design - eliminating contradictions and benefit from each others knowledge. For the customer, there is a potential economic benefit. Playing The Architectural Game might also save some unnecessary refactoring.

- *Improvement:* This is what The Architectural Game is all about. Playing The Architectural Game again lets the team redesign the application on a system-wide level - from worse to better.

- *Diversity:* Because The Architectural Game is played by everyone, everyone is able to contribute with their opinion on the design.

- *Opportunity:* The Architectural Game is an opportunity to make the team rethink the architecture, and make it adequate sustainable.

- *Redundancy:* The Architectural Game can and should be conducted whenever it is needed. Although the resulting architecture might be the same before and after, The Architectural Game is also used to clarify and strengthen some of the design decisions made by the team. Just because The Architectural Game doesn't produce any outcome, it isn't necessarily a waste of time.

- *Failure:* As mentioned under Redundancy, just because The Architectural Game doesn't produce any new material or perspective, it still clarifies the architecture. Therefore, this is in the spirit of XP, since failure in this case generates knowledge.

- *Accepted Responsibility:* Playing The Architectural Game is a responsibility every member of the team must take. It is not up to the team leader or manager to call for The Architectural Game, but up to the people doing the actual work - the pair programming. These are the ones who can see the potential problems in the current design of

the architecture and thus they are the ones who most obviously can call for it.

Conflicting:

- *Flow:* The Architectural Game might conflict with this practice. After all, when playing The Architectural Game the team must pause all activities thus not making it possible to keep a steady flow of completed user stories.
- *Baby Steps:* XP is all about taking small steps. The Architectural Game is also conflicting here as all activities are brought to a halt. Although XP prescribes that taking big steps is dangerous, The Architectural Game is taking a lot of baby steps towards an architecture which can support the current user stories - the customers requirements at the moment.

The principles of XP are guidelines for a team of developers through the development. The effects of augmenting XP (Beck, 2004) and choosing different practices than the ones originally embedded in XP (Robinson and Sharp, 2003) has already been studied, and as long as the new practices support the fundamental values of XP, it is not a problem. The Architectural Game does not conflict with the values of XP. It does, however, conflict with the principle of *Flow*. Playing The Architectural Game is, like any architectural activity, an investment in the future. It is not an activity that ends up with an exhaustive detailed design of the whole application and should be played with the evolutionary style of XP in mind. If the architecture isn't adequate sustainable in light of the known requirements, playing The Architectural Game will be a chance to improve the design. This ensures a higher quality, which in the end will produce a design with better fulfillment of quality criteria.

7 Discussion

Throughout this article we have proposed a new practice to XP: *The Architectural Game*. We believe that by introducing this practice to the XP practice portfolio, we increase the chance of the establishment of an adequate sustainable architecture. Thereby decreasing the necessity for unnecessary refactoring due to architectural issues.

Though we have made it possible that there is a need for this practice we acknowledge that following topics should be taken in account.

The introduction of The Architectural Game is based on empirical data collected from an XP project. This project only went through a single iteration. Based here upon it is hard to tell if a finalized XP project would have come up with a better architecture.

Furthermore, it could be said that the new practice interferes with the traditional XP practice flow. XP prescribes keeping the design simple, and implementing it as needed. Our new practice suggests stopping the project

implementation and doing the overall design. Although this might violate the simple design as well as the YAGNI concept, it is still carried out within a single iteration - thus only based on the user stories within this iteration.

The Architectural Game should be seen as an investment in the project. On the one hand an adequate sustainable architecture should be established early on thus eliminating the need for massive refactoring later on in the project. On the other hand if the customer change requirements to the system, the design might become obsolete and thereby a victim of the YAGNI concept.

7.1 Impact of revised description of XP

As mentioned in Section 2, the original description of XP was by Beck (2000). In 2004 Beck published an altered description of XP. The development project of our case study was developed using Steinberg and Palmer (2004). The applicability of The Architectural Game in XP was reviewed compared to the 2004 edition. Given the revision of the case study it is interesting to consider whether or not the result derived from the case study - namely that "XP does not always produce an adequate sustainable architecture" - still holds taking Beck (2004) into account.

In the next paragraphs, the "new" XP refers to the Beck (2004) version of XP, and the "old" XP refers to Beck (2000).

The values are in large the same in the new as in the old XP. In the new XP, the value *Respect* has been added. The development team in the case study was quite little - five developers. One could imagine that a lack of respect for the project would have manifested itself as an unwillingness to accept responsibility and to participate in the activities on equal terms. However, we can not find any evidence of such in the diary.

After a comparison of the principles in the new and old XP, we ended up with the following practices which have been added: *Self-similarity*, *Diversity* and *Redundancy*. The Self-similarity was already present in the old XP - writing tests that first fail and make the tests work again can be compared with an acceptance test. The teams were not put together using Diversity, but this wasn't a problem - or at least there was nothing in the diary suggesting this. The Architectural Game supports redundancy since it is an enabling practice to Incremental design.

Comparing the new core practices against the old practices, five stand out as having substantially new content. *Sit together, Whole Team, Ten-minute build*, and *Incremental design*. Since the development team worked in a single open space, the qualifications needed to solve the development problems at hand and that the build never took beyond ten minutes, *Sit together, Whole Team*, and *Ten-minute build* were all followed. During the course of the case study there were several refactorings, and the design decisions where based on the simple design practice from old XP. Considering the new practice incremental design, Beck extends the advice regarding design, and specifically writes:

> "*... Some of the teams who read and applied the first edition*

of this book didn't get the part of the message about the last responsible *moment. They piled story on story as quickly as possible with the least possible investment in design. Without daily attention to design, the cost of changes does skyrocket. The result is poorly designed, brittle, hard-to-change systems. ...* "

Following this new practice closely may have affected the outcome of the case study. However, the fact that Beck emphasizes the need for paying attention to design supports the need for some means of supporting the change of focus towards the design of the system. Our proposal to do that, is The Architectural Game.

Although XP has been revised, the result of the case study apparently is still valid with regards to the revised description. The development process was in accordance with the values and did not break fundamentally with any of the principles of the new description. However, Beck gives more attention to design with the practice Incremental design, which to some extent supports our introduction of The Architectural Game.

7.2 Quality Criteria

In section 4 we argued that *usability*, *flexibility*, and *comprehensibility* were in accordance with the four criteria for evaluating design from XP. However, some of the other criteria for architecture from Mathiassen *et al.* (2001) are for example not necessarily embodied by neither the four criteria nor the *simplicity* principle. *Security, efficiency, correctness*, and *portability* (Mathiassen *et al.*, 2001) are not addressed explicitly by the practices of XP. In a safety- or security-critical setting XP needs to be adapted to guarantee satisfaction of safety or security criteria (Beck, 2004). Further research may be conducted to establish whether or not The Architectural Game might prove as an enabling practice regarding satisfying these criteria.

8 Future Work

This section will outline proposed future work. In the article we have suggested a new practice to XP. Unfortunately, time did not allow another cycle of AR. This means that we can only speculate the outcome of a project with the new practice incorporated. This suggests a new AR project which uses The Architectural Game practice.

In Section 5 we say how often and when to play The Architectural Game. It should be clear that we haven't had the time to actually test the idea in depth. The prescribed "how" and "when" are only speculations. Therefore whether The Architectural Game should be a core practice or a corollary practice is up to future testing.

Moreover, an interesting aspect is the scope of requirements to take into consideration during a round of The Architectural Game. In this article we propose that the scope should be limited to the user stories for the current

iteration. However, it might prove usefull to extend the scope to all the known user stories and not only the ones selected by the costumer.

By playing The Architectural Game too often you might see that changes to the architecture will be done too often and take too much time - and using YAGNI this in direct conflict with XP. On the other hand using The Architectural Game as a corollary practice might mean that, when finally playing, the produced architecture will be the cause of major refactoring throughout the entire system - this might have been avoided if The Architectural Game was played more often.

This article does not differentiate systematically between architecture and design. However, an intuitive approach to a distinction between the two terms might originate in the abstraction level and the level of planning. Design being the least abstract (closest to actual implementation) and least planned, and architecture being the most abstract and most planned. Using these notions, XP considers design, whereas traditional methods also consider architecture. The Architectural Game thereby introduces architecture in XP, and thereby planning and a higher level of abstraction. A further study of the differences between the two terms and how they are used (or not used) in XP, may be interesting too .

References

Christopher Alexander. *Notes On The Synthesis Of Form*. President and Fellows of Harvard College, 1994.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. `http://agilemanifesto.org`, 2001.

Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2 edition, 2004.

Rachel Davies. The power of stories, July 29 2001.

Julio Cesar Sampaio do Prado Leite. Extreme requirements (xr). *Jornadas de Ingeniería de Requisitos Aplicada*, 2001.

Martin Fowler. Reducing coupling. *`http://www.martinfowler.com/ieeeSoftware/coupling.pdf`*, 2001.

Martin Fowler. Is design dead. *`http://www.martinfowler.com/articles/designDead.html`*, 2004.

Tom Gilb. *Software Metrics*. Winthrop Publishers Inc. (USA edition), 1988.

Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1. edition, 1999.

Leif Obel Jepsen, Lars Mathiassen, and Peter Axel Nielsen. Using diaries*. 1997.

Radmila Juric. Extreme programming and its development practices. *Information Technology Interfaces, 2000. ITI 2000. Proceedings of the 22nd International Conference on*, pages 97–104, 2000.

Jonna Kalermo and Jenni Rissanen. Agile software development in theory and practice. Master's thesis, University of Jyväskylä, 2002. `http://www.cs.jyu.fi/sb/Publications/KalermoRissanen_MastersThesis_060802.pdf`.

Craig Larman. *Applying UML and Patterns*. Prentice Hall, 3. edition, 2004.

Maria Carmen Leonardi and Julio Cesar Sampaio do Prado Leite. Using business rules in extreme requirements. *Advanced Information Systems Engineering*, 14(1):420–435, 2002.

Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objekt orienteret Analyse og Design*, volume 3. udgave. Forlaget Marko, 2001.

Judy McKay and Peter Marshall. The dual imperatives of action research. *Information Technology and People*, 14(1):46–59, 2001.

Hugh Robinson and Helen Sharp. Xp culture: Why the twelve practises both are and are not the most significant thing. 2003.

Jim Shore. Continuous design. *Software, IEEE*, 21(1):20–22, 2004.

Mary Beth Smrtic and Georges Grinstein. A case study in the use of extreme programming in an academic environment. Number 3134 in Lecture Notes in Computer Science, pages 175–182. Springer-Verlag, 2004.

Daniel H. Steinberg and Daniel W. Palmer. *Extreme Software Engineering: A Hands-On Approach*. Pearson/Prentice Hall, 2004.