From Pair Programming to Mob Programming to Mob Architecting

Carola Lilienthal(✉)

WPS – Workplace Solutions GmbH,
Hans-Henny-Jahnn-Weg 29, 22085 Hamburg, Germany
Carola.Lilienthal@wps.de

Abstract. The real life of a developer is not about development – it is about maintenance. Today typical programmers do not develop applications from scratch but they spend their time fixing, extending, modifying and enhancing existing applications. The biggest problem in their daily work is that over time maintenance mutates from structured programming to defensive programming: The code becomes too complex to be maintained. We put in code which we know is stupid from an architectural point of view but it is the only solution that will hopefully work. Maintenance becomes more and more difficult and expensive.

In this paper, I will show you how pair programming, mob programming and mob architecting help your team to avoid this apparently inevitable dead end. These three levels of quality improvement start with programming in pairs evolve to programming with the whole team (mob) and finally arrive at improving the architecture with the whole team.

Keywords: Software architecture · Teamwork · Pair programming · Mob programming · Quality improvement · Mob architecting

1 Introduction

Software systems are surely among the most complex constructions of humankind. It is hardly surprising that software projects tend to fail and legacy systems stay untouched because of the fear that they might fall apart when changed. The reasons why software development and maintenance fail can come from various levels: the application area and the organizations involved, the applied technologies, the domain fit of the software systems, or even the qualification of users and developers. In this article, I will focus on different techniques that can help development teams to reduce the technical debts of the software architecture. If the whole team is aware of the architecture and potential technical debts, the software architecture will stay maintainable and extendable with steady costs over a long period.

Let us begin with the typical path that the software architecture of a system takes and then discuss the techniques to prevent these dead ends.

© Springer International Publishing AG 2017
D. Winkler et al. (Eds.): SWQD 2017, LNBIP 269, pp. 3–12, 2017.
DOI: 10.1007/978-3-319-49421-0_1

1.1 From Structural to Defensive Programming

At the beginning of a software development project a team of experienced developers and architects will contribute their best experiences and their collective knowledge in order to design a durable architecture with a low level of technical debts. The term "technical debt" is a metaphor Ward Cunningham coined in 1992 [1]. Technical debts arise when developers and architects consciously or unconsciously make wrong or suboptimal technical decisions. Later, this wrong or suboptimal decisions lead to additional expenses and will delay maintenance and expansion.

Unfortunately, it is not enough to focus on this aim at the beginning of the project. According to the motto: We'll design a long-lasting architecture at the beginning and then everything is and remains good. On the contrary, a development team will only achieve a long living architecture if it constantly keeps an eye on the technical debts. In Fig. 1, you see what will happen when you permit the technical debts to increase over time or, which is much better, if the team is able to reduce them regularly [2].

Imagine a team that always evolves its system in releases or iterations. If we have a quality-conscious team in action, every member of the team will know that they add some new technical debt with each extension (yellow arrow in Fig. 1). During the extension, this team is therefore already thinking about how it could improve the architecture. Following the extension, the technical debt is then reduced again (green arrows in Fig. 1). A continuous sequence of extensions and improvement emerges. If the team follows this path, the system remains in a corridor of low technical debt.

If the team is not working in a steady sequence of renovations of the architecture, the system's architecture will slowly dissolve and maintainability deteriorates. Eventually the software system leaves the corridor of low technical debts (s. red rising arrows in Fig. 1).

The architecture has eroded more and more. Maintenance and extensions of the software are becoming more expensive to the point at which any change will be a painful effort. In Fig. 1 the red arrows are getting shorter and shorter indicating this circumstance. Per unit of time, the team will achieve less and less functionality, bug fixes and adaptations to other quality requirements due to this increasing architecture erosion. The team will become more and more frustrated and demotivated sinding

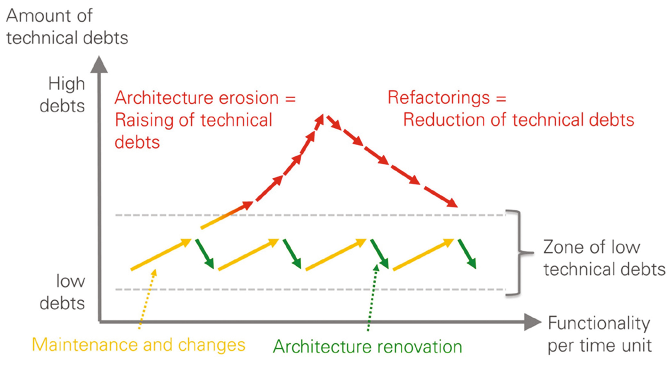
Amount of technical debts
High debts
low debts
Architecture erosion = Raising of technical debts
Refactorings = Reduction of technical debts
Zone of low technical debts
Functionality per time unit
Maintenance and changes
Architecture renovation

Fig. 1. Technical debts and architecture erosion (Color figure online)

desperate signals to project leadership and management. However, when management hears these warnings, it will usually be too late.

If a team is on the way up the red arrows, the sustainability of the software system decreases continually [3]. The software system will be prone to errors. The development team will be said of being cumbersome. Changes that could previously be done in two days, now will take twice or three times as long. Overall, everything will be going too slowly. "Slow" is in the IT industry a synonym for "too expensive". I agree! Technical debts accumulate and with every change, you have to pay the interest on the technical debts.

The path that leads out of this dilemma of technical debt is to improve the quality of architecture retroactively. This enables the team to gradually bring back their system into the corridor of fewer technical debts (see red arrows descending in Fig. 1). This path is arduous and costs money – therefore, it is a good idea to install techniques that keep a development team within the zone of low technical debts.

1.2 Uniform Structure and Guidelines

Software is in general hard to construct due to the immense number of elements needed in a software system. In my experience, a smart developer is able to overlook about 30,000 lines of code while changing the source code and anticipate the impact of a change in one place to the other parts of the code. Typically, operational software systems are much larger. Their sizes ranges from 200 thousand to 100 million lines of code [2].

These numbers make clear that developers need a software architecture that provides them with two things:

- A **uniform structure** that helps developers to find their way through the existing complexity
- **Guidelines** that restrict the design space and thereby give direction to the development.

If developers have an overview of the existing uniform structure, they probably will make changes to the software correctly. Guidelines will lead all members of the development team into the same direction, while unknown parts of the software are much easier to understand and comprehend. The software system will obtain its uniform structure. This uniform structure will be the basis of common understanding, so maintenance and enhancements will be quicker and more consistent.

Of course, the architects need to document the uniform structure and the guidelines. However, what may even be of higher importance is that all team members know and understand the structure and guidelines. To achieve this, knowledge must be spread throughout the whole team.

2 Spreading Knowledge in Teams

I frequently meet teams that keep their system's architecture under control despite its application area, its size or age. Enhancements and bug fixes are made in an acceptable timeframe. New developers can understand the source code with reasonable effort.

Why are these teams different? How do they manage to live in peace with their software architecture for a long time?

What I realize observing these teams is a heavy use of pair programming and mob programming. With our own teams, we added another technique: mob architecting. In the following, I will present these three techniques and describe the advantages and drawbacks of each of them.

2.1 Pair Programming

Kent Beck proposed pair programming in 1999 as a technique of extreme programming [4]. In pair programming, two programmers work together at one workstation. One, the pilot, has control over the keyboard and writes code. The other, the so-called navigator, reviews the work in progress and discusses his upcoming ideas for improvements and future problems with the chosen solution. The two programmers switch roles continually (Fig. 2).

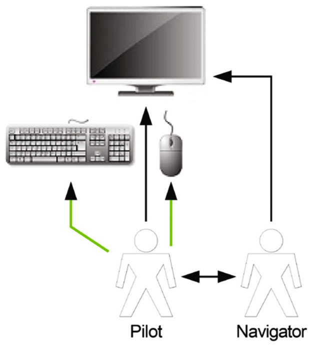

Fig. 2. Pair programming

Pair programming has been evaluated in various scientific experiments [5]. The quality of the code written in a pair is higher, with fewer bugs, better interfaces and less redundancy. Moreover, it is frequently reported that programmers have a higher confidence in their results and enjoy their work more with pair programming. As programmers report, this confidence and enjoyment mainly stems from the fact that as a pair they can work more focused and effectively. The time wasted by doing more than what is sufficient or unnecessarily switching tasks or being interrupted is noticeably reduced.

In our teams, we have equipped our pair programming workstations with two keyboards and two mice. This helps to prevent back pain, but poses a new problem: Pilot and navigator have to be very clear about who is coding. If not, the navigator will start annoying the pilot by interfering with his or her work (Fig. 3).

We have also done distributed pair programming. Two programmers, who were use to work with each other in one place now were located in two different places.

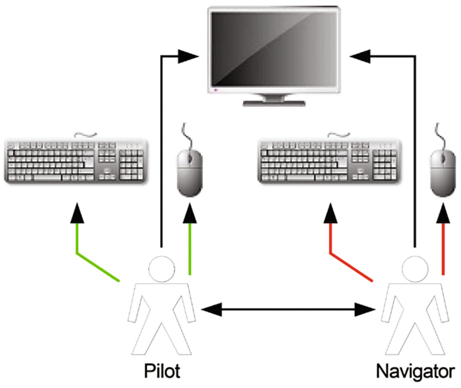

Fig. 3. Pair programming with double equipment

Nevertheless, they worked together sharing the screen and discussing their solutions via VoIP services. This solution showed very good effects on team building in distributed teams, but only worked in our experience if the members of the pair had been working together on-site before (Fig. 4).

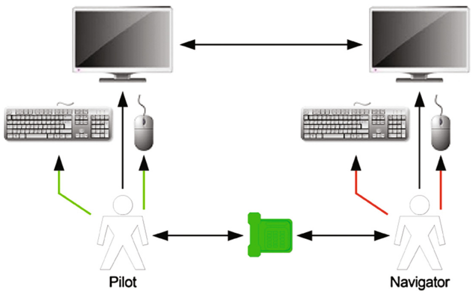

Fig. 4. Pair programming in different locations

Since the pilot in a pair has to explain his ideas and decisions to his counterpart their understanding of the architecture will continuously be challenged. Through programming in a pair, they will deepen and enrich their architectural knowledge of the whole system. Especially if pairs are switch repeatedly, the team members will distribute their knowledge continuously.

Spreading the knowledge and thereby easing incorporation of new team members or overcoming the loss of experienced team members is a big advantage of pair programming. However, the knowledge is distributed on a rather low level, such as by documentation in code. If the task of each pair is fixing a bug or implementing a new feature. The system's architecture will come rarely into focus. Maybe the pair will discuss the architecture but on a local level within the pair and they have no means to visualize the architecture while programming.

2.2 Mob Programming

Mob programming expands the idea of pair programming to an entire team. The whole team works together on one task, with one (active) keyboard and one large screen (usually a projector) at the same time. It is a kind of full-team pair programming [6].

Just as in pair programming, a pilot will use the keyboard and write the code while constantly explaining what he or she is doing. The rest of the team will observe the work in progress and discuss with the pilot upcoming ideas for improvements and potential problems with the solution chosen. As in pair programming, the pilot role is rotated regularly. There are different rotation strategies: ping-pong-pairing means that one pilot writes a failing test and then passes the keyboard to the next pilot who fulfills that test and writes a new failing one before passing on the keyboard. Timer pairing is another strategy where a timer organizes the rotation [6] (Fig. 5).

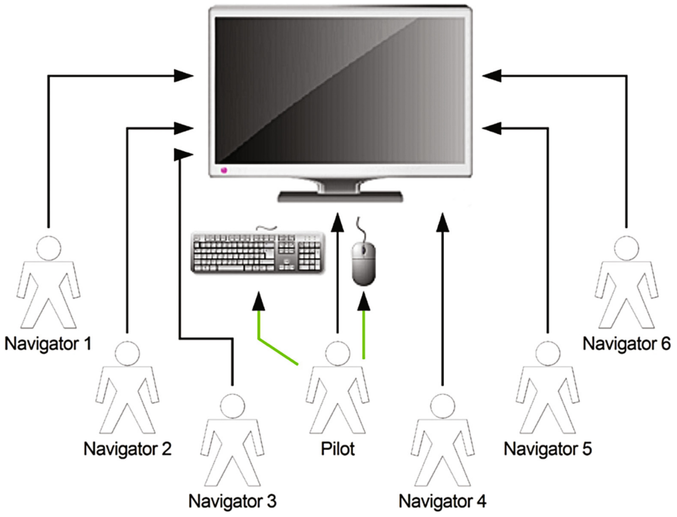

Fig. 5. Mob programming

Up to now, no scientific experiments have been conducted showing the usefulness of mob programming. Therefore, only experience reports of the advantages of mob programming exist. The teams using mob programming claim that they have achieved the maximal possible throughput for their team. Typing obviously is not the bottleneck in software development. The real cost factor is solving the problem with all the cycles of revision and rework typically involved [7].

Mob Programming addresses all these time-consuming issues. Working with all experts on one task is the best and quickest way to finish it. All the knowledge needed is available and the energy of the whole team focuses on one problem. This is the main advantage over pair programming, while all the other advantages of pair programming are also valid for mob programming: fewer bugs, higher quality in code and design and less redundancy.

In our company, we have one team who has started to practice mob programming in their daily work. They have reached the conclusion, that mob programming is the best solution for crucial new features and refactorings that will affect fundamental parts of the architecture. Our team calls this kind of changes “heart surgery”. The whole team in a mob always works on these far-reaching tasks. Their goal is to bring together the knowledge of the entire team and to spread the solutions and decisions within the team.

In summary, mob programming takes pair programming one-step further. But still a team only sees the architecture through the code. With mob architecting, this drawback can be finally overcome.

3 Mob Architecting

With mob architecting, the concept of mob programming is taken to a different level. Supported by a consultant and a tool to visualize the architecture the whole team can discuss and improve their system’s architecture in a very focussed and holistic manner. How does this work?

Initially, the tool for architecture visualization is loaded with the source code of the system. There are various tools available for architecture visualization and refactoring: for example Lattix, Sonargraph, Sotograph, Structure101. Since these tools are tools for experts, an eternal pilot who has a lot of experience in using the chosen tool typically drives a mob programming session [2].

The external pilot is conducting the development team through the process of mob-architecting (Step 1 to Step 4 in Fig. 6).

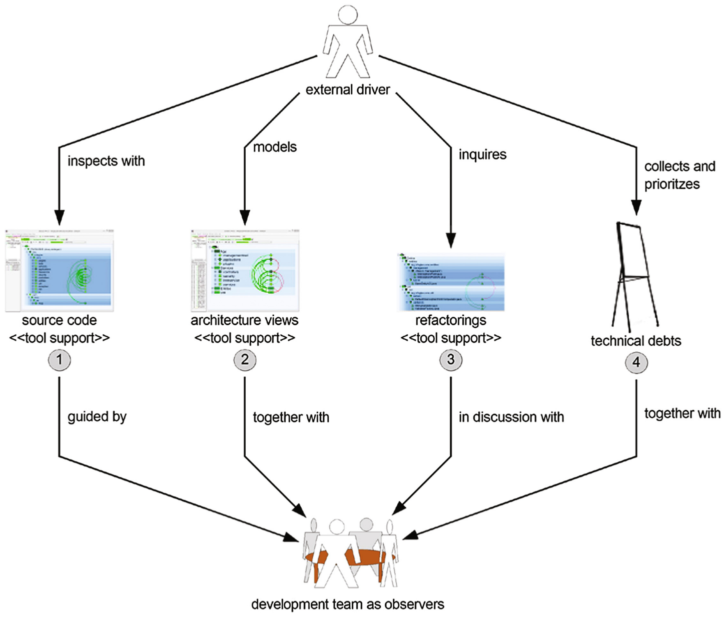

Fig. 6. Mob architecting with the whole team

3.1 Inspecting the Source Code (Step 1)

The first step is to analyze the source code in order to find the mapping of source code to architecture. Tools for architecture visualization can visualize the structures found within the source code and within the build-process tools, such as classes, directories, packages and build units. If the architecture is important for the development team, the main architecture should be detectable in these source code structures and build structures. Build units, packages, or directories should represent technical layers or domain modules.

In Fig. 7 (left side), the source code of an open source java system is presented by the tool Sotograph. The circles represent packages, the triangles represent files and the green arcs represent dependencies. Arcs on the left side of the vertical denote dependencies that go downwards. For example, within the package "plugin" there is a class that needs functionality from a class in the "net"-package. The arcs on the right side of the vertical denote dependencies that go upwards. For example, in the package "util" you will find one or more classes that need functionality from classes in the packages "entities", "controllers" and "security". All the circles (packages in this case) that are marked with a plus contain other packages or classes. The package at the bottom "com. frovi.ss.Tree" is opened completely, showing four classes belonging to this package [2].

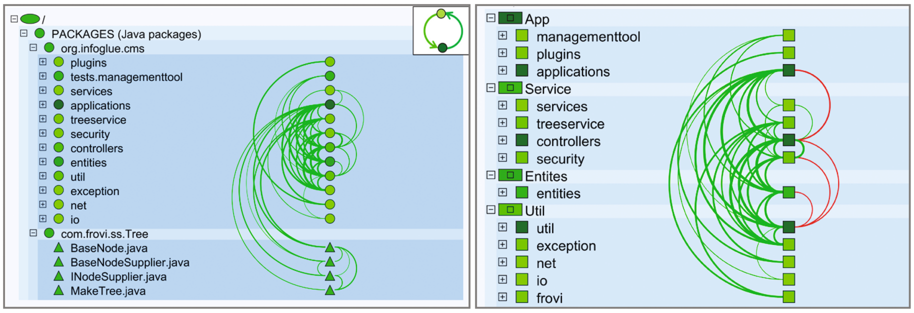

Fig. 7. Source code structure and modeled architecture

3.2 Modeling the Architecture (Step 2)

The second step of mob architecting (s. Fig. 6) is to model various architecture views onto the source code. On the right of Fig. 7, the architecture has been modeled onto the source code. In this case, the architecture is a technical layering composed of four layers "App", "Service", "Entities" and "Util". Within each layer, there a several modules. The layer "App" contains the modules "managementtool", "plugins" and "applications". The layers "Service" and "Util" consist of several modules as well, while "Entities" just has one module with the same name. These modules correspond nicely to the packages in the package tree on the left side of Fig. 7. This is not always the case and modeling the architecture (step 2 in Fig. 6) is time-consuming.

Figure 7 also shows that the three of the four technical layers, namely "App", "Service" and "Util" do not exist in the package tree. The knowledge about these layers exists only in the documentation or in the brains of some members of the development team. Missing matches like this will usually emerge during mob architecting. Then, the team will start to learn about their architecture. New members of the team will begin to catch the overall idea. Old members will be discussing their different ideas about the architecture.

3.3 Inquiring (Step 3) and Collecting (Step 4) Refactorings

In Fig. 7 (right side), some red arcs are shown. These arcs are depicted in red, because these dependencies violate the technical layering. Some classes from the module "util" in the layer "Utils" need some functionality from classes in "Entities" and "Service". Inspecting these red arcs and defining refactorings for these violations is the third step of mob architecting shown in Fig. 6.

The violation between the layer "Service" and "App" will serve as an example to understand how a violation can be refactored. Figure 8 left and right show a filtered view of the source code. Only the classes involved in the violation are visible. Therefore, only some packages are shown and not all their content is displayed.

In Fig. 8 (left side) you can see three classes from the package "org.infoglue. cms.controllers.kernel.impl.simple" that cause the violation (red arc) from the module "controllers" to the module "applications". They all call one method in the class "VisualFormatter.java". The line within the three controller classes perform the following call:

`value = new VisualFormatter().escapeHTML(value);`

Since the whole functionality of VisualFormatter is more like a utility than like an application functionality, the team decides that VisualFormatter should belong to the "Util" layer and not to the "App" layer.

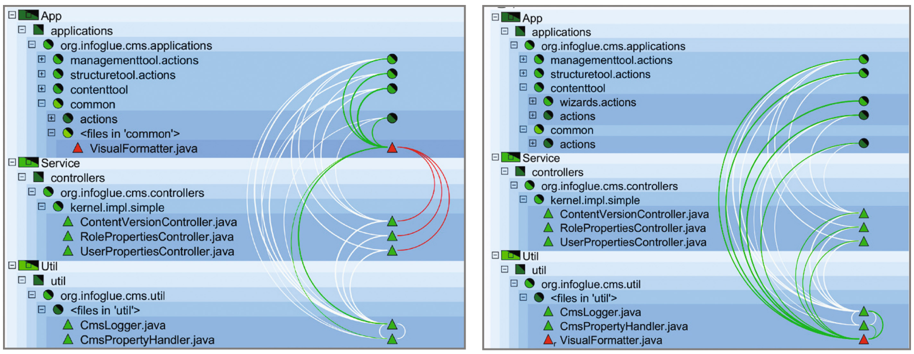

Fig. 8. Violation and refactoring for a very simple problem

The right side of Fig. 8 shows this refactoring. VisualFormatter has been moved to the "Util" layer and the violating dependencies are gone. Of cause, this refactoring is only done virtually in the tool and it is obviously a very simple refactoring.

No classes have to be redesigned and reimplemented. More complicated refactorings cannot be resolved by moving a class in the tool, but will lead to implementation work. As the final step 4 of the process of mob architecting (s. Fig. 6), the external pilot documents the refactoring on a flipchart or a whiteboard.

This process of modeling architecture, inspecting and collecting violations is an iterative process which is repeated several times during mob architecting. Not only the technical layering but also the segmentation of the system into domain modules and the use of patterns can be analyzed. Examinating different architectural views with the entire team stimulates discussion and helps the team inspecting their system quickly and on a high level.

Finally, the collected refactorings are prioritized and the improvements of the software architecture will be scheduled for the subsequent iterations.

4 Summary

In this paper, I have shown how pair programming, mob programming and mob architecting can help your team keeping their system's architecture in good shape without technical debts. Three levels of quality improvements have been presented and the value of each of these techniques have been discussed. Applying these techniques to development teams can help them improving their overall understanding of their software architecture and spark architectural discussions. More information on the various techniques of mob architecting can be found in my book "Langlebige Softwarearchitekturen" an English version is in preparation.

References

1. Cunningham, W.: The WyCash portfolio management system. Experience report, OOPSLA 1992 (1992)
2. Lilienthal, C.: Langlebige Softwarearchitektur, Technische Schulden analysieren, begrenzen und abbauen, dpunkt.verlag (2015). English version in preparation
3. Martin, R.C.: Principle and Patterns (2000). http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
4. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston (1999)
5. Succi, G., Marchesi, M.: Extreme Programming Examined (XP). Addison-Wesley, Boston (2001)
6. Jansson, P.: Get a good start with mob programming. https://thecuriousdeveloper.com/2013/09/15/get-a-good-start-with-mob-programming/. Accessed 25 Aug 2016
7. Obermüller, K., Campbell, J.: Mob Programming - the Good, the Bad and the Great. http://underthehood.meltwater.com/blog/2016/06/01/mob-programming/. Accessed 29 Aug 2016