Prototypes are forever
Evolving from a prototype project to a full-featured system

Hugo Corbucci[1] and Mariana V. Bravo[1] and Alexandre Freire da Silva[1]and Fernando Freire da Silva[1]

Agilbits, Sao Paulo, Brazil,
{hugo,marivb,freire,fernando}@agilbits.com.br

Abstract. Prototypes are a well known, widely accepted development practice but, if not carefully evolved, they can become a nightmare to maintain. This paper presents the experience of a four person agile team who successfully grew a prototyped system to a full-featured application without any clear transition in the project. The paper describes how the project started with a very simple prototyping goal, evolved through iterations and spikes to a partly working system and transformed, in the end, in a complete application widely tested and refactored.

Key words: prototype, agile methods, refactoring

1 Introduction

Prototyping is an activity that most developers have heard about. Fred Brooks mentioned it in The Mythical Man-Month [1] as one of the best ways to provide a quick view of a feature to the clients or users to help them make a choice. Many agile methods - Dynamic System Development Method (DSDM) [2] is heavily based on prototyping - adopt ideas related to this concept, like spikes [3] in Extreme Programming.

Successful software prototypes look very much like complete features given a certain execution path. Therefore it is common that the customers are so satisfied that they want to integrate the prototypes into the working system and move on. The problem is that prototypes are frequently created in a "quick and dirty" fashion and the result is not adequate to be incorporated in a full-featured system. Furthermore, it is quite hard to explain this fact to the stakeholders who usually do not want to invest any more money in this "already working" feature. The consequence is that they switch priorities, focus developer work efforts on other parts of the system and leave the rough prototype lost within the code base. In the long run, the prototype becomes a part of the system but it is filled with bugs, unhandled corner cases and, frequently, cruddy code. Nobody remembers what it was supposed to do or whether it is really important. Maintainability is deeply affected and developers get that natural and unpleasant I-told-you-so feeling.

Developers who have been through the pain of maintaining dirty prototypes are no longer enthusiastic about prototypes. If they have to, they arrange their work so that there will be absolutely no way to integrate the prototype to the existing system. They often do this by either using a different platform, language or even creating prototypes in other media. This inflexibility can reduce the ability to respond to changes quickly and therefore harm the clients' interests.

This paper describes how a four-people collocated team managed to start a prototyping project and evolve it naturally into a full-featured application. We have organized this experience report in chronological order of the project's evolution. Section 2 will present the project as it was first introduced to the development team. Section 3 presents the work process established by the team to create the software based on prototypes. After some time, the team felt that the customer needed a full featured application. We describe this change in Section 4. The following section (Section 5) shows how the team adapted to evolve the prototype to a production ready application. Finally, Section 6 presents the current status of the project and Section 7 concludes with a summary of practices that we used to go through this experience without much pain.

2 Starting the project

Back in March 2008, our company was hired to do some consulting for one of the largest movie production companies in Brazil. The client had a great idea for an application for movie-script writers but had absolutely no knowledge about software development. He wanted to investigate the idea and understand how much investment it would take to turn it into working software in order to establish his business plan. The company's job at the time was to scout the market, discover competitors and provide an estimation of the work required to develop the client's idea.

For this, a consultant was assigned the task of understanding the client's needs and desires and two developers were asked to analyze the existing script writing programs and evaluate possible development paths. After about 3 weeks of consulting and study, the team produced a deck of story cards with two estimates each, based on the use of two possible platforms. The first platform was an existing open source software with several features and a copy-left license[1]; the second one was an Eclipse Rich Client Application[2] developed from scratch using Eclipse's open source framework.

This initial estimate suggested that a four person team dedicating four daily work hours would be able to build a working prototype of each feature in approximately nine months using the existing open source software and in roughly one year using Eclipse's platform. The open source solution had the advantage of providing full functionality of several other features. For a complete system,

[1] http://celtx.com/ – Last accessed on 27/02/2010
[2] http://www.eclipse.org/rcp – Last accessed on 27/02/2010

the estimate was well over two years of work on the Eclipse version and about a year and a half for the open source one.

After some discussion, the client opted for the Eclipse based solution due to a license restriction of the open source one which conflicted with his business plan. He also chose to develop only a prototype of the idea since two years seemed like too heavy of an investment for him.

After the exploration phase, the consulting contract ended and a new negotiable scope contract [3] with emphasis on development effort was signed. This new contract established a team of four developers working with an open scope that would be negotiated monthly, providing 160 hours of work each month. It specifically stated that the developers would work in pairs all the time and that the developed system should have automated tests for the production code.

The features initially presented to the team were organized by the client into three priority groups. The first group contained all features most critical to the client, the ones that would allow him to experiment with his "big idea". The second one consisted of some features already present in most script editors in the market, such as editing the script text itself. Most features in this second group were in fact epics [4]. The third group contained only the features needed to read and export to different file formats, such as those used by competing programs.

The project's goal was to create a visual high fidelity prototype with mostly faked or simplified features from the first group. The client would use this prototype to present his ideas to investors by October 2008. This meeting would either boost the project's development to a full featured system if the investors liked the idea or end its development in the case that they rejected it.

This was the team's vision of the project when development began, a short seven month project whose fate would be decided by its capacity to impress investors. Therefore, the main goal was to provide support for the client's demonstration to ensure the project's growth and success. The next section (Section 3) describes how the team organized itself to achieve this goal.

3 Prototyping phase

Given the project's objective, the customer always prioritized new features considering only one specific usage scenario. This meant that, for most features, there were several cases which the team was asked **not** to handle. Regarding the source code, it meant that no verification or validation was written and the prototype would likely crash if the user did not behave as expected. We also incorporated several spikes as permanent solutions and did not handle a fair number of exceptions.

The team knew from the start that the client would change his mind over time. After all, it was partly to better understand his idea and its applicability that he wanted to build this prototype. This meant that features would be developed and later thrown away while code produced only for a quick spike was going to become part of the system. Therefore, the team invested only as much

design, automated tests and refactoring as needed to keep the system flexible enough to receive the next changes. The team also made it clear to the client that some work would need to be done on features after he accepted them in order to polish the work.

From the start, the team installed a continuous integration server[3] to automatically build a new version of the system hourly. With this in place, the client could follow the system's evolution, test the features and provide feedback within a very short time span. This allowed for absolutely no surprises in review meetings and greatly improved the team's ability to tune each feature as it was released.

The first few iterations went quite smoothly, developing features from the first phase which involved importing a script in a text-only format marked with some meta-data and providing a simple way to manipulate and visualize this data. For these first features, it was easy to avoid inconsistencies since there were few business rules involved.

Meanwhile, the client's demonstration script was evolving as the prototype did and the team was able to use conditionals as needed to ignore cases he would not enter during his presentation to investors. By October 2008, the main features from the first group were ready for demonstration, although not finished and polished for real use. However, by testing and playing with these features, the client felt that the program lacked an important aspect of script editing programs: the text editor itself. It was an epic initially included only in the second group of features, but he wanted to see how the text editor would integrate to allow viewing and editing of the meta-data he was creating. Therefore, he prioritized the inclusion of an incipient text editor in the prototype and started to detail stories related to this editor.

As a result of this discovery, the client did not feel completely confident to present the software to the investors; even so, he started to make contact with a few people to schedule a meeting during December 2008. That date became our new deadline. All efforts were focused on making a prototyped text editor available for the demonstration.

At this point, the pressure for polishing the new features increased. Time was running out for the demonstration that would decide the project's fate. The customer wanted the team to ignore corner cases, speed up delivery and ensure the demonstration would run smoothly. The excitement from this important presentation to other people (only our client and us had seen the software so far) was strong motivation for the development team to deliver all features the client had asked for. Yet, despite unit testing and pair programming being mandatory rules, the general will to quickly deliver the features decreased the code quality considerably. However, external interference was about to change the situation. Section 4 will explain how the project got affected and what new direction those changes pointed to.

[3] http://www.jetbrains.com/teamcity/ - Last accessed on 27/02/2010

4 Changing the rules

December 2008 came and went without any meeting. The company that the client was in contact with had just been acquired by another one so any project presentation was useless until things settled down. This relieved the pressure of the upcoming deadline and we finally acknowledged the burden of unhandled technical debt. All members of the development team agreed that the code was getting complex and the quality was decreasing. This had a negative effect on productivity and speed which complemented the client's request for a more complete text editor, with fewer mock-ups and simplifications. The first iteration of 2009 was dedicated entirely to refactoring the prototyped editor we had so far, and the iterations that followed further developed this editor to include more business rules.

Although the meeting with investors was mentioned in this period, again it did not take place. The general feeling the development team had was that the project was no longer aimed at a simple presentation to investors. It was slowly becoming a more elaborate end-user oriented application. This seemed to be confirmed by two important and concomitant changes that happened between June and July 2009.

In that period, the client was asking for a much more complex and full-featured text editor. At the same time, he was exploring and wanted to develop new features to evolve his previous idea. At this point, the client started to understand the dilemma that the developers had felt so far. How could we keep a good rhythm of new feature delivery and still cover most of the use cases for current features?

The team estimated that it would take at least three full iterations to have an editor with the more complex functionality expected by the client. He did not welcome this news, since it would mean that no new features would be added for a while and he wanted to explore them with the investors' meeting still in mind. We then decided to investigate other possible solutions for the text editor rather than just developing from scratch. After some research we discovered an open source Eclipse Rich Client WYSIWYG (What You See Is What You Get) HTML editor[4]. The editor relies on a reimplementation of Eclipse's `StyledText` component which is responsible for rendering text within Eclipse editors. This project was close enough to the one we needed to implement, having some of the functionality our client wanted on our application, and it was maintained mostly by one Russian company. We suggested the possibility of outsourcing the development of the underlying text editor infra-structure to this company, thus enabling us to continue working on the new features the client wanted. He accepted this idea and this was the first change that confirmed the shift towards a more full-featured system.

At the same time, the team learned that the client had formed a dramaturgy experts group to help him better understand how to structure the application, the existing features and the new ones. The software was now going to have a set

[4] `http://onpositive.com/richtext/` – Last accessed on 20/02/2010

of beta testers and it needed to perform decently to allow the users to suggest improvements. However, the current development approach would not be able to support this new use of the system. The change had to be clear to the client so that development efforts would be directed to address this new way of working.

Evidence of the software's deficiency came quickly from the dramaturgy study group. They started having trouble with several known and unknown corner cases, unexpected behaviors and just plain old bugs. The client recognized this problem and, despite the fact that it would mean new features delivered more slowly, decided it was time to invest more in usability and user experience. This was the second change that confirmed the shift in the project's purpose.

Meanwhile, the development team was concerned with the increasing complexity so they started to track some data from the source code. The first metrics were the amount of `FIXME`, `TODO` and `XXX` marks in the code. Since the beginning, the development team had added those marks everywhere they felt a corner case or a behavior existed but was not handled. Each mark had a small comment associated and the kind of the mark determined the criticality of the problem. Figure 1 shows the evolution of those marks during the project.

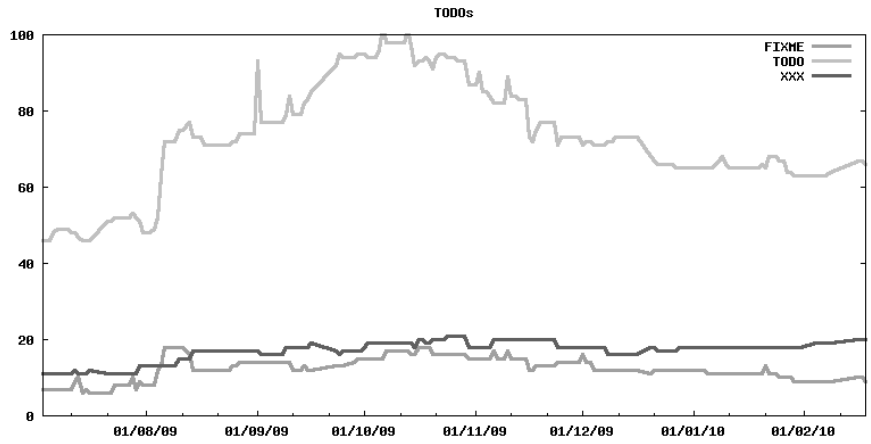

Fig. 1. Evolution of FIXME, TODO and XXX marks in the source code

It is noticeable that the first data collection of those marks is dated for mid July 2009. It took the team some time to consider this metric was important enough to be automatically tracked. To consciously acknowledge that the team needed to track complexity was step one to adapt to the new direction. The next section will present other practices that allowed us to handle this shift.

5 Adapting to the new rules

The amount of `TODO` marks was fairly high but the team still had to develop new prototyped features and therefore they decided to just track it and try to keep it under control. While this gave the team some idea of code complexity, it did not help to show if this complexity was being tamed by tests.

By the beginning of August 2009, the client decided that it was important for him to be able to see the evolution of the work done by the Russian team and he asked to have two editors available on the application. The first one was our original simplified one and the second one was based on the new outsourced component. At this point, we introduced a lot of code duplication because features available in the original editor were supposed to be also available on the new editor.

The result was a huge increase in the marks tracked as well as an increase in lines of code. The team suspected tests were not following this trend. A very simple script was written to count the lines of production code and test code. Figure 2 shows how both metrics evolved over time. It starts at the same date as Figure 1 to facilitate comparisons.

As with the `TODO` metric, the lines of code showed the team a small issue regarding testing but it was not addressed immediately. It was well known that the code did not have much coverage. There were no User Interface (UI) tests and quite a few features were related to the way data was shown. However, the team had a feeling that the effort to test the controller and model should have produced more test than code.

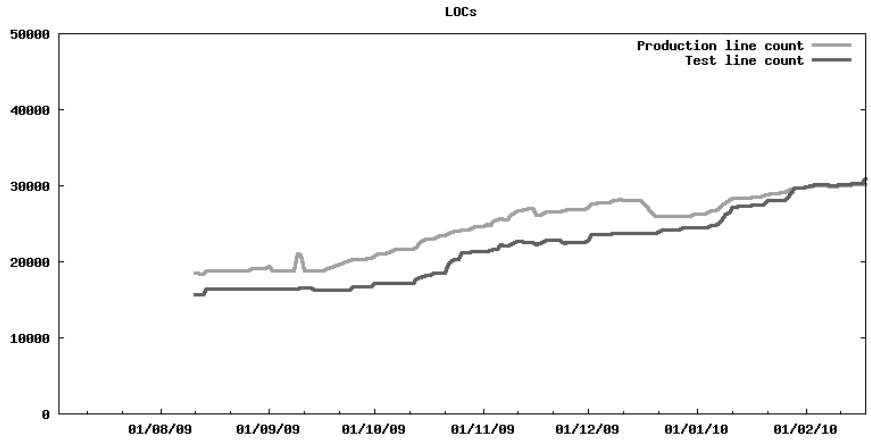

Fig. 2. Evolution of production lines of code and test lines of code

By this time, the customer brought up the idea of a presentation to the investors again, the project had been going for quite some time and he was feeling he had spent enough money and needed some external investment. Therefore the

old investor meeting pressure appeared within the team and the deadline was, again, the end of the year. The goal was to quickly integrate the outsourced component and tune a few features for the much expected meeting.

Because of the rush to deliver new features, the code had reached a critical situation. `TODO` marks were amazingly high and distance between tests and code was at its highest level. During this period, the work was being roughly divided in three tasks: fixing bugs, implementing new features and integrating the outsourced component. Around September 2009, the team also suffered the loss of a member who was required for full time consulting work. The team went from two pairs to one pair and an extra person, and the velocity decreased.

Figure 3 shows yet another complexity metric used by the team. Counting the number of occurrences of the `instanceof` Java reserved word led the team to understand the level of special cases unfactored in the code. The team felt reducing this number would lead to better factored code.

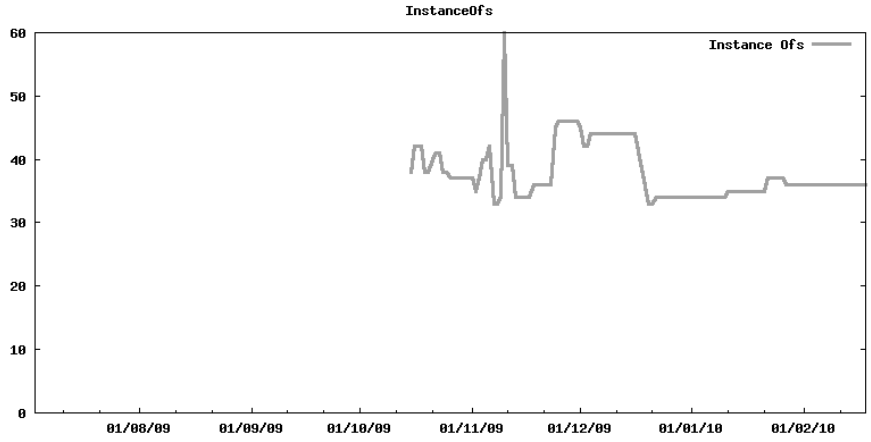

Fig. 3. Evolution of instanceof in the code

Curiously, as with the two previous metrics, the effect of tracking the number of `instanceof` was not immediate. More interestingly than that, in a short time span, having the metric did not stop it from getting worse. In the rush to produce the features, the team was becoming reckless with code quality. Something had to be done or the project was going to become very hard to maintain and the client's demonstration would surely suffer from it.

With three persons working at the end of the year to match two pairs velocity and reducing complexity, the team had to calm down, step back and rethink about agile values. Section 6 presents the resolutions the team adopted and their impact on the current version of the system.

6 Current status

December arrived and it was time to perform a full integration of the outsourced component and throw away the team's first solution. Along with this came a considerable decrease in `TODO` marks, `instanceof`s and lines of production code. A large source of duplication was erased. By that time, the team was mostly working with just two persons pairing full time since the third developer was involved in other projects and rarely managed to pair up with the others.

The team decided to profit from this fact and instituted a merciless refactoring policy. No matter how small or how big the refactoring was, it had to be done and it was to be included as a regular part of each task and not as a separated task. Corner cases were not to be left unhandled and the execution of any user behavior which the client had not prioritized was to be logged to the application error log. The impact of those policies was fairly clear by the end of the month. All metrics had improved and more bugs were being identified by the development team.

The month passed and no meeting was scheduled. The famous deadline was, once again, a myth. With code quality increasing, `TODO` marks decreasing, test code improving and bugs being caught by the development team (instead of the users), the team felt an extra developer would help increase velocity and so started to train one to join the team in January. The general feeling was that the client was getting ready to wrap up the project since he was quite happy with the software and was considering he had invested too much already on this idea.

However, in the next meeting, the client presented an officially hired beta tester that was going to support the team in order to improve the usability of the system. He also presented the team with many new features and usability improvements that he wanted to have done. The client also mentioned that he was seriously thinking about dropping the quest for investors and releasing the product by himself. In order to do so, bugs needed to be eliminated. All bugs found were immediately given the highest priority and should be solved in some way as soon as possible.

Since that time, the beta tester has helped the client to identify some inconsistencies in the business rules. Thanks to the experience that the client had accumulated so far in the project, he allowed himself to experiment with a few solutions and alternatives to these problems until he felt more comfortable with the way features worked, allowing him to seek consistency, conceptual integrity and the certainty that other choices were not as good as the one he had made.

7 Conclusion

Successful software prototypes need not always be completely thrown away. This fact comes from the very nature of software and the ability to easily and quickly merge pieces of code together to produce a working program using agile method-

ologies. Agility relies on this fact to allow for incremental evolution of the system and, therefore, supports that fact that prototypes can grow to complete features.

By embracing the idea that, if your prototypes are successful, they will be incorporated into the software, you can be prepared to maintain that code. It does not mean that prototypes should be developed exactly as well known complete features. Prototyping should allow you to explore different solutions and as they survive the selection process they should be refactored and evolved in a similar fashion to the set-based lean approach [5].

Refactoring, testing and decoupling are essential to allow code evolution and should be practices that are strongly enforced as the prototype starts to transform into a working feature. Unhandled exceptions, cases or behaviors should be documented with tests or some other mean that allows for quick listing and search.

Although prototypes are not first class production code and much of their value comes from this lower class status, they are likely to turn into essential features of the system. It is very important to track this evolution and provide the necessary support to maintain quality. Even so, there should be no fear about throwing away code either. The value provided by prototypes resides deeply in the knowledge they bring to the customers, which means the code is much less valuable then we give it credit for. In our experience, using prototypes to experiment with different ideas for the same feature proved invaluable, and many accepted prototypes were able to evolve to production code in the application.

References

1. Brooks Jr., F.P.: The Mythical Man Month: Essays on Software Engineering. Addison-Wesley (1975)
2. DSDM Consortium: DSDM: Business Focused Development. Addison-Wesley (2002)
3. Kent Beck and Cynthia Andres: Extreme Programming Explained: Embrace Change, 2nd Edition. Addison-Wesley (2004)
4. Cohn, M. : User Stories Applied: For Agile Software Development. Adding-Wesley (2004)
5. Poppendieck, M. and Poppendieck, T.: Implementing Lean Software Devleopment: From Concept to Cash. Addison-Wesley (2009)