Divide *After* You Conquer: An Agile Software Development Practice for Large Projects

Ahmed Elshamy[1] and Amr Elssamadisy[2]

[1] ThoughtWorks Inc.
Chicago, IL 60661 USA
Aselshamy@ThoughtWorks.com
[2] Valtech Technologies
Addison, TX 75001 USA
Amr@Elssamadisy.com

Abstract. Large software development projects are not agile by nature. Large projects are not easy to implement, they are even harder to implement using agile methodologies. Based on over 6 years of experience building software systems using agile methodologies we found that we can modify agile methodologies to be successfully applied to large projects. In this paper, we will introduce a development practice, which we call *Divide After You Conquer* to reduce some of the challenges during the development of large agile projects. By solving the base problem first with a smaller development team (*Conquer phase*) before expanding the team to its full size (*Divide phase*) we can solve many of the problems that occur with larger projects using agile methodologies.

1 Introduction

Large software development projects have their own set of problems that need to be addressed [2,3,4,6,8,9]. Roughly speaking, we consider a development project *large* if the development team is anywhere between 50 and 100 people (includes developers, testers, business analysts, and managers). Many of the standard development practices in agile methodologies do not provide their expected consequences [1,2,9].

In this paper we describe a development practice that we have used on several different projects at multiple companies. This development practice, which we name 'Divide After You Conquer', solves many of the base problems first before expanding the team to its full size. This practice is related to much work that has been done before in non-agile development processes [6,7,10] – i.e. this is not a new problem. These practices and processes include prototyping, architecture-driven development, and a full upfront high level design as recommended by the Unified Process to name just a few. *Divide After You Conquer* is however different from each of these practices in that it is permanent and not throw-away like prototypes and is done in a test-driven manner as opposed to upfront design as suggested by the Unified Process [6].

2 Challenges in Applying Agile to Large Projects

One of the aspects common to many agile development methodologies is that the entire team (business analysts, developers, and testers) collaborate very heavily. With a large project, this type of collaboration is difficult at best. What we have found again and again that we tend to break up into subteams for better communication. The downside to subteams is the possibility that the subteams build stove-piped subsystems if communication is insufficient among the teams. Even if the group communication is successful we can have problems of consistency and duplication that goes undiscovered. Of course, there are other practices that help alleviate these problems such as rotating team members among the subteams, having an overall design document, etc. [4,6]. We are not invalidating these techniques, but in our experience they are not sufficient to alleviate the problems typical when separate subteams build separate parts of the system. Another way to state this problem is that the different subteams may result in a non-homogeneous and inconsistent architecture.

Also, as we have indicated above, large projects using agile methodologies may not be as amenable to recognizing and responding to change. Specifically there are two different aspects, i.e. the recognition of a change and the response to that change.

2.1 Recognizing Change

The first part, of recognition of change, is greatly affected by the size of the team and the size of the artifacts (code, use cases, tests). As we have more people, whether or not we have multiple subteams, it is more difficult to determine if a change in one part of the system affects other parts of the system. The standard way that this has generally been addressed is either upfront design to make sure that everything matches and extensive documentation. With typical agile development practices upfront design is looked down upon because of the cost of design carry and the fact that requirements change. Agile development methodologies also tend to be light on documentation, and non-agile methodologies that are not documentation-light have documentation that frequently is out of synch with the project.

2.2 Responding to Change

The second part, responding to change, is usually done via refactoring. Refactoring is a good solution that relies on a large test framework as a safety net. There is nothing wrong with this, refactoring is very efficient in general. There are, however, large refactorings which are difficult and expensive to perform – so we want to minimize these refactorings. We have the non-agile solution to this problem which is design upfront, but designing upfront generally results in a design that is more complex than that needed by the exact system causing a design carry cost throughout the lifetime of the project. This particular problem, that of upfront design, has been discussed extensively in the agile development community. We need to find another way to solve our large refactoring problem other than upfront design.

3 Divide After You Conquer

Basically, instead of dividing the work first and then solving each sub-problem, the starting team is a core team (usually about 20-30% of full team) that has the most experienced developers, testers, and business analysts and it builds out the main business cases in a test-driven manner. This first phase lasts a non-trivial amount because we want to build out enough of the project that we touch all of the primary business areas (without dealing with alternative/exceptional scenarios) and build out most of the architecture. Because we have a small team, a full agile methodology works without modification. We end the first phase when we have a stable code base with a significant portion of the architecture built out and a broad swathe of business built. At this point we have *conquered* the problem and now it is time to *divide* by growing the development team and splitting up into smaller subteams to grow the project into a fully functional software system. Because the architecture has been built out in a test-driven manner we have the amount of complexity needed but no more. Teams now have a homogeneous architecture in the different subprojects. We want to stress that this is not the *only* practice needed for agile development with large projects, but it is a significant one. Unlike [2] we clearly define when we reach a stable architecture. [2] recommends just declaring the architecture is stable to give the courage for developers to work on the existing code.

3.1 The Conquer Phase

The conquer phase of the project will introduce a stable working example of the architecture and system design. This working example is built iteratively with constant refactoring and ensures that the design works for the current requirements. The system design may include layering, object models and screen layouts. All that would be a starting seeds for other subteams to follow and build upon. Creating the design through an iterative process according to the business need reduces the risk of redesigning the system when a standard upfront design approach is used.

The team will define a set of use cases that are broad enough to touch/interact with most parts of the proposed system. These use cases have to be useful to the business and simple enough to be implemented within a few months. The goal of the conquer phase is to implement these use cases in an iterative and a test-driven manner.

Testers and business analysts will come up with standards/working examples for story cards, testing criteria, testing framework that would be followed after the split. This work will act as the basis for later work by the larger team. This can be seen by many as ‘reinventing the wheel’ and that these standards can be set upfront. We found out that building a set of experiences for this specific project that can be reused by the larger team in the divide phase is more effective than reinventing the agile development process for each sub-project.

The initial development team (developers, testers, business analysts) that started on that early project they would have a very good over-all picture of the project. They were involved in the implementation of the simple business cases that touch most if not all business areas. They also understand the overall flow of the application. This knowledge will enable a better split into teams when we reach the divide phase. This group will also work as mentors for the remaining team in the next phase.

Finally, the starting project is not a prototype. It is a set of production quality stories, code and tests that will be used as the basis for building out the entire system. In this case the conquer phase has the same goals as the elaboration phase in a development process like the Unified Process [10] – namely to flush out the architecture and address any high-risk areas. Some problems may not appear unless we try to implement a real life situation with a complicated enough business use case.

3.2 The Divide Phase

Divide Subteams by Business Areas: The divide phase starts when the project is divided into sub-teams, which is a common practice for large projects. Sub-teams are more manageable and they can adapt to change more easily. A common practice also is to split based on business areas. Dividing into business areas helps understanding the business within a business area – understanding the business is of prime importance to a successful project. The business functionality will also drive the code, which is a major benefit of applying agile methodologies. Jutta Eckstein [2] shows similar practice. Eckstein recommends starting small and growing slowly. It did not emphasize on when to start dividing into subteams or when to start to grow the team. We clearly recommend splitting the teams after completion of the broad business case and the team will only grow after the division into subteams.

Staff gradually: Staffing the sub-teams would be by assigning members of the existing team (the team on the conquer phase) to sub-teams. New team members will also be assigned to sub-teams. No specific requirement on the newly joined members, except being open for using agile methodologies. The staffing may occur as initial staffing to start the sub-teams and gradually add new team members to sub-teams as needed. Staffing gradually would help the knowledge transfer to be done smoothly, releasing the load on the conquer team to do knowledge transfer properly to new team members. The initial staffing should allow for pairing between members of the conquer phase and newly joined members. Pairing may include developers, BA and QA team members as well.

Transfer Knowledge: In the beginning of the *Divide Phase* knowledge needs to be transferred from the core team members to the additional team members. There are several ways that we have seen this done: code reviews, pair programming, and mentoring are three of the most common techniques. Code reviews are not necessarily one-shot deals but can be done repeatedly until the expanded team becomes cohesive. Pair programming is not always the easiest practice to implement depending on the environment of the company, but when allowed has been one of the most successful techniques we have seen. Finally, mentoring lies somewhere between the two extremes where the core team members take on the role of mentors to the new members to transfer business, testing, and design knowledge.

Rotate team members from the newly staffed members: During the course of project rotation of team members between sub team will be helpful to transfer knowledge between sub-teams. Members rotated to new teams may work on interfaces between the two sub-teams, the one they were originally on and the newly joined. This also helps maintain the consistent and homogenous architecture that we built in the *Conquer* phase. By rotating the team members they are exposed to the entire system which allows for a large project version of the eXtreme Programming practice of *Collective Ownership*.

4 Challenges in Applying This Practice

The practice mandates having a highly skilled set of developers at the beginning of the project (the conquer phase). The high skilled developers should be available for the rest of the project. In the beginning of the divide phase staffing with other developers, business analysts and testers should start. In some situation this staffing pattern may not be applicable, due to some organizational structure of the company. Still in some other companies as it's the natural way to staff a large project. Companies will give much attention to large projects and they would staff them with their best team members in the beginning. As the team grows they may staff the project from new hires, consulting companies or developers that are freed from other projects. This common scenario may match the staffing time line proposed.

Defining the use cases or the core part of the system is the main challenge when applying this practice. Some systems may have convoluted set of functionality with high interaction. Finding a simple business case that satisfies the core system is hard is such systems.

Knowledge transfer between subteams is still a challenge. Members' rotation is still not enough to ensure successful communication between subteams. Other practices must be involved to enhance the communication between subteams and ensure proper interfaces between teams. These practices like [8] are out of scope of this paper.

References

1. Cockburn, A. Agile Software Development, Pearson Education, Indianapolis, IN. (2002).
2. Eckstein, J. Agile Software Development in the Large: Diving into the Deep, Dorset House Publishing, New York, NY. (2004).
3. Elssamadisy, A. XP on a Large Project- A Developer's View, in Extreme Programming Perspectives, eds. Marchesi et al., Pearson Education, Indianapolis, IN. (2003).
4. Elssamadisy, A. and Schalliol, G., Recognizing and Responding to "Bad Smells" in Extreme Programming, presented in International Conference on Software Engineering 2002.
5. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software, Pearson Education, Indianapolis, IN. (2004).
6. Jacobi C. and Rumpe, B., Hierarchical XP: Improving XP for Large-Scale Reorganization Processes, in Extreme Programming Examined, eds. Succi et al., Pearson Education, Indianapolis, IN. (2001).
7. Larman, C. Applying UML And Patterns, Prentice Hall, Upper Saddle River, NJ. (2001).
8. Rogers, O., Scaling Continuous Integration, presented in XP 2004, (2004).
9. Schalliol, G, Challenges for Analysts on a Large XP Project, in Extreme Programming Perspectives, eds. Marchesi et al., Pearson Education, Indianapolis, IN. (2003).
10. Scott, K. The Unified Process Explained, Pearson Education, Indianapolis, IN. (2001).