# How Extreme does Extreme Programming Have to be? Adapting XP Practices to Large-scale Projects

Lan Cao
*Georgia State University*
*lcao@cis.gsu.edu*

Kannan Mohan
*Baruch College*
*kannan_mohan@baruch.cuny.edu*

Peng Xu, Balasubramaniam Ramesh
*Georgia State University*
*pxu/bramesh@cis.gsu.edu*

## Abstract

*The need to develop software at Internet speed and accommodate changes during the entire software development life cycle has made lightweight or agile development methodologies like Extreme Programming popular. However, such methodologies have been considered to be beneficial for small and medium sized projects, with small teams. In this research, based on a case study in an organization that develops large-scale, complex software using a modified form of extreme programming, we highlight the key differences between agile principles proposed in prior literature and the agile practices that are suitable for large-scale, complex software development. Based on these differences, we propose general guidelines on tailoring agile development methodologies to make them suitable for the development of large, complex software systems .*

## 1. Introduction

Organizations increasingly face a rapidly changing business and technological environment. In such a dynamic environment, traditional software development approaches which assume that all requirements can be anticipated at the beginning of projects and will remain stable are unlikely to be successful. The inability to continuously evolve the software to be in alignment with changing requirements means being unresponsive to business conditions, which leads to business failure [10]. Agile software development approaches such as Extreme Programming (XP), Crystal methods, Lean Development, Scrum, and Adaptive Software Development (ASD), have been proposed to deal with today's turbulent business and technological environment.

The goal of Agile Methods is to allow an organization to be able to deliver quickly and change quickly. [1, 5, 10]. The proposed agile practices in software development vary, but they share the common characteristics such as iterative development, working in frequent consultation with the customer, and having small and frequent releases. [1] One of the widely used agile methodologies is Extreme Programming (XP), a software development approach that advocates rapid iterations, rigorously tested code and working closely with end users [3, 4]. XP has been successfully adopted in some small software projects [13, 17].

Proponents of XP claim that using this method brings advantages over conventional processes, including lower management overhead, higher team productivity, happier customers and shorter release cycles. However the applicability of agile approaches is constrained by several factors such as project size and type, experience level of project personnel, and access to committed customers. Boehm [5] argues that agile methods are difficult to scale up to large projects because of the lack of sufficient architecture planning, over-focusing on early results and low levels of test coverage. It is also recommended that agile methods not be used in mission-critical software development. However, in the current dynamic business environments, agility is also needed for large projects that face the same issues addressed by agile methodologies such as changing environment, ambiguous user requirements, and time pressure. Software development organizations are under intense pressure to deliver products rapidly, but with high levels of quality assurance. Approaches that can deliver just agility or are only plan-driven focusing on quality can't meet these needs. The ability to achieve and quality is required now more than ever [5].

Agile methods cannot be adopted directly for large, complex projects due to the lack of up-front

design and documentation [15]. However, most experts agree that agile methodology and traditional approaches are philosophically compatible [16]. For example, XP practices have been mapped to SW-CMM model that usually is considered appropriate for large-scale projects or organizations [14]. Efforts have been made to tailor XP methodology for large, complex projects to achieve faster development cycle times [8, 16]. Besides these case studies, principals or guidelines on how to adopt agile methodology for large-scale, complex projects have not been proposed in the literature. In this research, based on a case study in an organization that develops large-scale, complex software using a modified form of extreme programming, we identify agile practices that are suitable for large-scale, complex software development. These practices are also compared with agile principles proposed in prior literature. Based on the comparison, we argue that these agile practices provide a general guideline on tailoring agile development methodologies to make them suitable for large and complex projects.

## 2. Agile software development methodologies and large-scale projects

Traditional approaches emphasize the importance of project plans and documentations, trying to control unexpected changes. However, major changes in requirements, scope and technology are out of the control of development teams. The question often is not how to minimize changes in a project but how to better handle inevitable changes throughout its life cycle [10]. Agile methods respond to this expectation by adopting strategies designed to reduce the cost of change throughout a project.

The highest priority in agile methodologies is satisfying the customer through early and continuous delivery of valuable software [1]. Agile methodologies accommodate changing requirements, and anticipate them even late in development. Strategies such as delivering working software frequently, reliance on face-to-face communication, closely working with customers, and keeping designs simple, are used to mitigate the risks caused by changes. These methodologies emphasize a fair amount of planning. More value on the planning *process* is placed than the resulting *documentation* [5]. The tacit knowledge embodied in development teams is heavily relied on. These methods work best when team size is small and customers are dedicated to projects.

Several agile methodologies have been developed with the above objectives. They include Extreme Programming (XP), Crystal methods, Lean Development, Scrum, Adaptive Software Development (ASD). Among these, Extreme Programming (XP) is the most widely used. XP is a lightweight methodology that dispenses with much of the usual application development process, such as lengthy requirements definition and extensive documentation. It emphasizes on keeping development teams small and the code simple [4]. The XP life cycle has four basic activities: coding, testing, listening, and designing [14]. XP improves a software project in four essential ways: communication, simplicity, feedback, and courage. It encourages continual communications with customers and teams, maintaining simplicity, providing frequent feedback via testing, and dealing with problems proactively [14]. Twelve core practices are proposed to achieve these goals [1].

XP practices focus on maximizing communication and enhancing team-work. Managers, customers, and developers are all part of a team dedicated to delivering quality software. Communication barriers between developers and customers are removed by having customers work with developers onsite. Daily stand-up meetings and pair programming enhance project communication among team members, while lowering overhead. Extensive communication and quick feedback help build trust between customers and developers [12].

XP also empowers developers by enhancing their sense of project control [13]. By adopting XP, developers know where their project is heading and whether it is delayed. Furthermore, constant testing makes the programmers more aware of how well the code meets its expected functionalities. This knowledge improved the programmers' motivation.

Though software projects can benefit from agile methodologies, not all projects can directly adopt them. Characteristics of large, complex projects make it difficult to use agile methodologies directly. Large-scale, complex systems face three major issues: the thin spread of application domain knowledge, fluctuating and conflicting requirements, and communication and coordination breakdowns [7]. First, deep-application knowledge from development team, required by large-scale, complex projects, is thinly spread through many software development staff. Such distributed knowledge must be integrated. Substantial design effort has to be spent in coordinating a common understanding of both the application domain and of how the system should perform. Second, the lack of application knowledge of

[1] http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html

developers and a variety of events such as changing business goals and policies may cause fluctuating and conflicting requirements in large, complex projects. Third, a large number of groups have to coordinate their activities and share information during software development. Extensive effort needs to be spent defining terms, coordinating representational conventions, and creating channels for the flow of information.

Agile methodologies lack up-front design and investment in life-cycle architecture, and rely primarily on tacit knowledge of individuals and informal communication. All these practices can cause high risks. For example, teams may make irrecoverable architectural mistakes due to the lack of appropriate design. Customers to whom the developers have easy access to may not have enough knowledge of the requirements. Informal communication may not be effective when dealing with a large number of stakeholders and vast amounts of information that are characteristic of large projects. Traditional software development approaches, that are plan-driven, reduce these risks by a variety of strategies such as investing in life-cycle architecture, documenting necessary details, using formal external reviews, relying on strong leadership etc.

However, large-scale, complex projects also face dynamic change in project requirements and time-to-market pressure. Studies show that agile methods can be mapped to SW-CMM models, which is considered a traditional guidance model for software development process [14, 16]. To mitigate the risks in time-to-market and changing requirements, efforts have been made to adopt XP in large-scale projects, with mixed results. Elssamadisy [8] and Reifer [16] report on the large projects that try to adopt the XP methodology. These studies show that some practices from agile methodologies, such as iterative development, frequent testing and feedback, small release, and refactoring, are suitable for large projects. However, some practices, such as standing-meetings and the use of metaphors to describe system architectures, are not appropriate. It is found that without an overall design, it is difficult to maintain a big picture of the project as the systems grow. The communication between developers is also problematic when a large number of stakeholders are included. Current literature does not provide any guidance on how to tailor agile methodologies for large-scale, complex projects. In this research, a case study is conducted, from which a set of practices on how to adopt an agile methodology for large projects is proposed.

# 3. Agile practices for large-scale, complex Projects

We are currently conducting a case study with a large software development organization (called FinApp hereafter) developing large-scale, complex corporate financial applications. In this paper, we present initial insights gained from this study.

## 3.1. Case description

FinApp is involved in the development of a complex enterprise system that provides financial services to banks, insurance companies, loans and brokerages. The project includes more than 1,000 business objects in six different categories. There are 22 developers working on this project. The project team started with a modified XP approach. Specifically, a subset of XP practices has been modified for use in this project. The project had successfully delivered several applications before FinApp was merged with another large organization.

## 3.2. Need for agile approach

FinApp had recognized the key role that certain XP practices could play, when modified appropriately, in developing their complex financial systems. The sensitive and mission critical nature of the enterprise system, in the views of critical stakeholders, well defined and enforced its processes. However, there is growing recognition that lack of agility makes it difficult to make changes to accommodate evolving requirements. Also, solely relying on formal communication causes problems in understanding and communication within the development team.

In the following sections, we discuss how FinApp has tailored XP practices to suit their development environment, and how these practices conform to agile principles described in the literature.

## 3.3. Tailoring XP for large and complex project

The business domain for FinApp is relatively mature and stable. The application, by nature requires high level of reliability, security, as well as quality. *"The difference is what we got here is an enterprise application and that is what drove everything… We have an extremely modified version of XP exactly because we have enterprise application",* says a project manager at FinApp. The enterprise application requires a sound abstract design that can be applied to

different applications. The system architects spent six months in developing business architecture and system infrastructure.

Finapp is a "huge" application, with 7,000 program files all together. Beyond the size, the application involved corporate cash management, which is a very complex domain. Security is critical and quality is the focus of development effort. Even though users requirements change over time there is a basic pattern for business objects in the application domain. Based on these considerations and restrictions of the project, FinApp took a cautious agile approach by tailoring XP practices.

**3.3.1. Designing up front.** XP de-emphasizes up front design because it is claimed that everything is changing. Instead, a "metaphor" is used to describe the basic elements and relationships of the application (Beck 2000). However, for complex, large-scale projects upfront architectural design is considered to be essential. In the FinApp project, upfront architectural design is both possible and necessary.

First, the banking environment is relative stable and the system can be described by a few typical business patterns. The project manager states, *"...So it turns out that there is a fundamental pattern for this part, there is a fundamental pattern for this part, there is a fundamental pattern for this part. It turns out there are 6 or 8 patterns that covers most of the system".*

Second, for complex and mission critical applications like a banking system, the use of best practices such as flexible architectures and design patterns is important. Upfront stable architectural design results in a strong backbone that could support several services built on top of it.

For a mission-critical application, upfront design makes sure that the system meets security and reliability requirements. "*They (the customers) have to make sure they don't have a pension plan built on any one of them, you know, so it has to be totally secure, totally documented and very clear from upfront*". Customers need to ensure that the developers have not built a feature into the system to siphon out or divert money. So all the design and source code are delivered to the banks.

Upfront architectural design reduces the development time for new functionalities. Each new functionally is based on the infrastructure backbone and design patterns drawn from Gamma et al [9]. With this approach, the time and effort for implementing new functionalities are much reduced when compared to developing from scratch. "*… each time we get a new service …, what I am doing is mapping the pattern. There is no need to technical specs. …we don't have to do detail technical specs because we were following established patterns."* FinAPP project established new services based on the existing services and design patterns. For example, an application can be readily configured to satisfy clients in the finance industry in different countries, instead of developing custom solutions.

Early architectural design also reduces developers training time thereby mitigating the costs of brining someone new on board. A project manager at FinApp comments: "*What I can do, on any given day, is to unfold all the teams and put new people in them, I can have people who start on Monday and be productive on Thursday. Where most organizations take six months I can actually have… because you talk the bank pattern in a day and a half, and on Thursday you can create your own based on the patterns*"

Upfront design helps control the costs of change. For large projects, the changes made to the system in response to change requests or for fixing errors cause unintended consequences and weaken the system over time. Design patterns are used to control the damage of changes by mechanisms such as centralized exceptional handling.

Upfront design provides the developers with a clearer understanding of the entire system and helps them understand how several services can fit into the backbone architecture. XP assumes that most stories are independent [4], however, it was found to be untrue in large and complex projects. Taber and Fowler [19] also reported that, as a system grows, it becomes more and more difficult to maintain a big picture of all the implemented functionality and how they all worked together. Architectural design becomes even more important for complex, large-scale applications because developers and analysts miss the connections and dependencies between stories as the application size and complexity grow [18]. An alternative to this could be having a design that evolves with the process instead of a complete upfront backbone design [11]. Such evolving design is seen to have the problem of tight coupling between the persistence layer and business layer, which makes it difficult to refactor the code after several months of development. FinaApp avoided this problem by having a detailed, upfront, stable architecture. It uses stable and standard interfaces, and patterns across the board. With the infrastructure backbone, new services can just be plugged in, following established design patterns.

***3.3.2.*** **Short release cycles with layered approach.** Short (2 to 4 week) release cycles and continuous integration are touted as one of the highlights of agile software development methodologies like extreme programming. For large-scale and complex projects,

this is still seen as a relevant principle, with adaptations. "*Because everything is ambiguous… What do customers want? Well, if you give me exactly what they want, they are gong to look at it and say, that's neat, can you do this, I want something different. I mean programming change is normal, adding the sophisticated understanding and ambiguity is inherent in the situation… So to me, you have to have this collaborative on-going rapid development cycle, short release."*

The upfront structural design of the backbone typically takes longer. At FinApp, the architectural design took around 6 months. The layered approach, similar to XP short release practices, delivered end-to end functionalities that can go to production at the end of each short iteration. This practice differs from XP in that the duration of the iterations are not fixed, but are based on the nature of the layers and tasks.

A FinApp project manager, commenting on the need for this adaptation, states: *"If you are doing layered deliverables, if we are late, we still got some stuff that works, rather than to say I don't have anything that works*." The focus is on getting production code that supports functionalities end-to-end rather than developing heavily integrated modules.

**3.3.3. Surrogate customer engagement.** The ideal customer for XP is an on-site customer who is also an end-user and has the ability, knowledge and courage for making decisions. Customer involvement is a key factor for XP project success. However, in reality, such access to customers is often difficult. In large-scale projects, the problem is amplified as the complexity of the application domain is often beyond the experience or expertise of a small number of customers as well as the developers. The scope of software development expands to include a variety of stakeholders. The composition of development teams becomes more diverse, involving users and management from throughout the organization. Another problem is that accessible customers are often not the end users of the system.

At FinApp, the real customers are banks and the end users of banks or other financial organizations. The customers are not readily available to the development team. Though it is difficult to have an ideal customer involved, the degree of customer involvement is considered very important for development at FinApp. *"We don't have access to a real customer. So a true onsite customer and the stories, they weren't an option. "*, says a project manager at FinApp. The customers are surrogated by product managers or business analysts, who have direct contacts with customers. Product managers meet with the development team almost everyday to discuss changes in requirements and to make decisions on development options. Such a practice led to better project outcomes when compared to projects that had infrequent customer engagement. *"The most successful case I had was with automated clearing where I actually got the product manager to attend the meeting every morning at 10 am …. Any developer who had an issue or a problem attended. So what we did there was negotiated, problem-solved, modified the requirements, modified the spec, there really was a collaborative XP type of process where in the consultation with our onsite customer in this case. So that worked out great",* says the project manager at FinApp. There is a clear need for continuous customer engagement in large-scale, complex projects. When direct access to customers is not feasible, the use of surrogate customers who are domain experts appears a reasonable compromise.

**3.3.4. Flexible pair programming.** Although pair programming is seen as a good practice, it is not considered to be realistic in all situations [8]. In FinaApp, pair programming is used in a flexible way. Only analysis, design, test case development, and unit testing are done in pairs. Developers do try to code in pairs and most of them return to solo coding. This is inherently tied to the personality of the developers. The level of pairing is customized to suit to the nature of the developers and the nature of the tasks that they perform. A project manager at FinApp comments on this need to tailor the practices to suit the organizational culture as follows, *"I am not going to insist people share the monitor. That's just dictatorial. I guess a lot of the developers that I hired here have hacker personnel. They really want to code."* This issue of flexible pairing is intimately related to motivating the developers and keeping their morale high. Sharing the same monitor and keyboard is also seen as a flexible aspect that works better when left as a choice to the developers.

The benefits of pair programming are also appropriately recognized. Pair programming is seen to:

- Reduce development time - Developers in pairs work faster as reported in FinApp Project. According to a project manager, two developers working together in a project finishes the project in roughly 80% time that would have taken if they were assigned separately. *"I can say that with confidence because I have scheduled them both way"*. The communication between two developers makes the communication documentation and management unnecessary.
- Reduce training time - The developers learn from each other since developers with different level of experience and skills are paired together. *"You are*

*always mentoring or being mentored"*. "*what I can do, on any given day, is to unfold all the teams and put new people in them, I can have people who start on Monday and be productive on Thursday. Where most organizations take six month",* claims a FinApp project manager.

- Improve quality of code - Communication across developers is seen as an essential factor that reduces inconsistencies and defects during development. A project manager at FinApp says, *"Because there's really synergy and one of the biggest things is the communication where the two developers have to convince each other that they know what they are going to build before they build them. And what the typical developer does is half understands an idea and is trying to work it out. Whereas if you actually have to write the test cases first and explain it to somebody and get somebody to buy it, they'll pick up the inconsistencies and the holes. That to me is probably the biggest plus in the whole thing."* This also relates to the issue of collective ownership. Pair programming increases the knowledge spread across developers in the team, thereby enabling any developer to tackle almost any problem related to the system. Continuous rotation of developer pairs facilitates such collective ownership. *"It was task oriented, but I pretty much made sure that I arranged the tasks so that nobody will, no team will work together for more than 6 weeks and there was an average of 2 week", s*ays a project manager. Such rotation facilitates developer learning and enhances the sense of collective ownership.
- Create a collaborative and supportive environment - Developers always have somebody to look for help in a paired setting. Moreover, these pairs work with teams of pairs and there's always a network of people to talk to and learn from. By doing this, a supportive and collaborative environment is created.

**3.3.5. Identifying and managing developers.** FinApp considers the hiring process to be highly related to the success of its agile practices. Developers who can recognize the importance of pattern-based development and providing standard interfaces across the system are seen as key to successfully projects. Developers' knowledge about architectural design and design patterns is seen as critical. Approximately only 15% of the applicants are considered to be qualified for this project environment.

Upholding developer morale is also seen as key to successful project outcomes. At FinApp, it is believed that motivated developers are better performers. Hence, incentive schemes and flexibility initiatives focus on motivating developers. *"It created an environment we have zero turnover"*. The FinApp project manager proceeds to comment that: *"…Respecting their contribution, and not reducing them to cogs on the machine. And I see what happens when you are going to say to reduce your developers to just cogs. They are terrible. They are unhappy and the only people you get are bad people"*. Flexible working hours, remote working, focus on results rather than micro-management, are seen as key factors that affect developer morale and motivation.

**3.3.6. Reuse with forward refactoring.** Pattern-based and interface-based development approach facilitates reuse to a considerable extent. Refactoring is seen as a key technique to enhance reuse across functionalities. However, refactoring here may take on form different from what is defined in XP practices. In XP, refactoring is a practice that improves the design of the code without changing the functionalities of the program. Repeated code is removed, code is reorganized and cleaned, common lines of code are abstracted out into separated classes. In FinApp project, instead of changing existing code, "forward refactoring" is largely used. "Forward refactoring" is an approach to develop new features by reusing existing code instead of developing new solutions. Existing services are untouched, new services are developed based on existing ones. For example, a project manager in FinApp comments: *"So what we did was starting with the account transfer that already been built. We refactor forward. What we did was we took the account transfer and say 'ok now we are going to build wire transfer… You built it in XP, you built once to satisfy your current need, then you get a new set of requirements, you overlaid them and find out that these are the part that stays the same. This is the part that changes. So this is where I have to insert my design patterns in order to isolate change from static"*. Existing design and code is refactored forward to include new functions. In summary, in typical XP practices, refactoring is a way to improve the design and making the system more robust, in FinApp, refactoring is seen as a technique for reuse with the support of the upfront architectural design. The architecture is designed in such a manner that it is not restricted to just banks. It can be tailored to be used in any financial institution.

***3.3.7.*** **Controlled empowerment --organizational structure.** Project managers at FinApp consider deep hierarchical organizational structure to result in unresponsive environments with high inertia. A project manager at FinApp notes: *"What we had in*

*terms of XP, first of all, we didn't have layers and layers of management. I just hate those layers of middle management. It is the worst thing about our occupation. We got rid of those…we got an extremely fast, extremely responsive, extremely flexible environment".* Decentralizing development-oriented decision-making is seen as critical for a successful agile practice. *"From my point of view, the concept of XP, a lot of them come from OO management --push decision making down, to invoke productivities, empower the people who are actually doing the work. Part of that was: not to insist that they share a monitor."*

The empowerment is controlled in an elegant manner through the committed use of standard interfaces and design patterns. The developers cannot stray away from a standard approach to solving the problems at hand.

### 3.4. Beyond XP practices - interesting findings

In summary, at FinApp, concept of patterns and up front design, a layered deliverables approach, and pair programming are seen as the most important practices for developing large-scale, complex applications. In addition, our study identifies some interesting practices that are different from but not necessarily inconsistent with XP principles.

**3.4.1. Impact of up front design on other practices.** According to XP practices, big upfront design is not appropriate to embrace frequent changes. However, it is interesting to find that in this case that upfront architectural design actually supports other agile practices such as pair programming, refactoring and short release.

The benefits of pair programming are realized with the support of design patterns. The patterns predefined the structure of the system and foster understanding of the whole project. Based on the deep understanding and standard patterns, communication between developers is more efficient. Further, design patterns also eliminate the need for detailed documentation and layered management. This helps reduce the development time and supports rapid development. Most importantly, the knowledge transfer between pairs is largely enhanced by the design patterns. New people are always paired with someone who knows the patterns. A project manager put it this way: *"…I can have (new) people who start on Monday and be productive on Thursday … because you talk the bank pattern in a day and a half, and on Thursday you can create your own based on the patterns. … two weeks we can cover all patterns."*

The benefit of reuse by refactoring in FinApp is also supported by the design patterns. In FinApp, since all functions are developed followed the design patterns, existing functions are largely reused to build new functions. "*To me, XP and design patterns are all the same things, and refactoring."*

**3.4.2. Management support and organizational culture.** The FinApp case shows that management support is critical for successful adoption of an agile software development approach. FinApp has been very successful in delivering quality products on schedule and within budget until the company merged with another big traditional banking company. The new organization values structured procedures, formal and detailed documentation and hierarchical management more than agility. "… *there are these concepts that if I have enough procedures, If I have details on my procedures, then everything is going to be taken care of. We don't have to worry about it. But it is simply not true that at certain point, the procedures get so heavy weight.*" The change in organizational culture resulted in the termination of the use of agile approach. Detailed documentation was required. The attempted reversal from a lightweight to heavyweight methodology is resulting in lack of communication among the stakeholders and thereby increases number of defects due to misinterpretation.

## 4. Agile Principles vs. Practices for Large-scale and Complex Software Development

Based on a discovery colloquium, Baskerville et al [2] have described a set of agile development principles (Table 2). Also, based on a study of software development practices in multiple U.S. Internet software development companies, they have identified a set of Internet speed development practices. They compare the agile principles to traditional software development *principles*, and with Internet speed software development *practices*. They observe that *'each of the Internet speed development practices come together to enact each of the agile principles'*. They also compare the practices and principles that were identified by their study with the principles proposed in the agile manifesto (www.agilealliance.com), which characterizes the values of agile methods and how agile methods distinguish themselves from traditional methods.

Modified XP practices, while successful at FinApp, are aimed at providing agility within the constraints of a large project. These practices still are guided by the agile principles. Table 3 shows the mapping between

Modified XP practices followed to develop large-scale, complex projects and agile principles that are proposed by Baskerville et al. [2].

**Table 2: Agile Principles (Adopted from Baskerville et al 2003)**

| No. | Agile Principles |
|---|---|
| 1 | Accept multiple valid approaches |
| 2 | Accommodate requirements change |
| 3 | Engage the customer |
| 4 | Build on successful experience |
| 5 | Develop good teamwork |
| 6 | Effective software development conforms to project environment constraints |
| 7 | Prepare for unexpected consequences from innovation in software processes |

**Table 3: Comparing Modified XP Practices in Large Projects and Agile Principles**

| Agile Practices for Complex, Large-scale projects | Agile Principles |
|---|---|
| Designing upfront | 1, 4, 6 |
| Short release cycles with layered approach | 2 |
| Surrogate customer engagement | 3 |
| Flexible pair programming | 1, 5 |
| Identifying and managing developers | 4, 5 |
| Reuse with refactoring | 4 |
| Flatter hierarchies with controlled empowerment | 7 |

From table 3 we can see that most of the agile practices identified in FinApp case fit well with the agile principles. This match, on one hand, illustrates the appropriateness of these practices. On the other hand, it also suggests the appropriateness of following agile principles in large complex projects.

**Practice 1: Designing upfront.** Instead of adopting the whole set of XP practices, FinApp used a "modified XP" approach. It combines designing upfront with agile practices such as short release, pair programming, and refactoring. This practice follows ***principle 1*** "accept multiple valid approaches". This practice matches with ***principle 6*** too. Use of multiple approaches, on the other hand, results from constraints of the environment. FinApp is large in scope, complex in functionality and mission critical in nature. Development of this kind of system requires a carefully designed architecture.

**Practice 2: Short Release cycles with a layered approach.** FinApp delivers end-to-end functionalities in each short iteration. By doing that, the system continuously accommodates requirements changes (***Principle2***). The delivered functionalities suit the customer need rather than focusing on documenting detailed specifications.

**Practice 3: Surrogate customer engagement.** This practice is a modified version of an XP practice -- on site customer. This practice follows the agile ***principle 3*** "Engage the customer". FinApp cannot access its real customer, instead, it uses product managers as a surrogate.

**Practice 4: Flexible pair programming.** Following agile ***principle 5*** which is to "develop good team work", flexible pair programming is used in FinApp. This practice is a modified version of XP "Pair programming" practice. Contrary to "always paired" in XP, in FinApp, developers are paired in analysis, design and testing. Coding is done by solo programming. The combination of solo programming and pair programming (***Principle 1***) overcomes some shortfalls of pair programming (e.g., developer's resistance), while still benefiting from it where feasible.

**Practice 5: Identifying and managing developers.** People factor is more important in agile development than in traditional development [6]. FinApp team emphasized choosing the right people for the team and created a collaborative environment to support teamwork (***Principle 5***). Developers' knowledge and experiences on different aspects of a project are greatly valued (***principle 4***).

**Practice 6: Reuse with forward refactoring.** This practice maps to the principle of building on successful experience. Refactoring is used as a technique to enhance reuse. Agile methodologies like XP do not emphasize development for reuse. They place a priority on speed, responsiveness and improvisation [2]. Developers usually focus only on their current need instead of building components for later reuse. However, for a large project like FinApp, development of upfront architectural design and use of design patterns are critical. Functionalities of the system are developed based on design patterns. Also, modules that have been developed to handle specific functionalities are refactored and made generic enough so that they can be tailored to handle different functionalities. By doing this, code can be reused to speed up development. This approach shows that reuse does not actually conflict with the agile approach and should be included in agile principles for large-scale projects.

**Practice 7: Flatter hierarchies with controlled empowerment.** In FinApp, layers of management are

replaced by a slimmer organization that improves communications between stakeholders and increases productivity. Developers are empowered to make their own decisions. On the other side, for a large and mission-critical application, the empowerment might cause unexpected consequences such as incompatibilities among the development process and products produced by different developers. . FinApp took several steps to deal with this issue (***Principle 7***). The upfront architecture design and design patterns are used to standardize development. For example, exception handling is centralized to control potential damages resulting from changes that are necessary. No agile principle deals organizational structures. However, it is found in our study that organizational issues are very important to the successful adoption of agile approaches.

## 5. Conclusion

Given the need to develop software at Internet speed, to accommodate changes in requirements, lightweight methodologies are becoming increasingly important. However, such methodologies cannot be applied readily to every project. They have to be tailored to suit the nature of the system and the development environment. This is especially true for large-scale, complex, and enterprise systems. Based on the initial findings from a case study, we have proposed a set of agile practices that have been tailored to be suitable for large-scale, complex projects. We have compared these tailored practices with agile principles that were proposed prior research. Creating a stable architectural design upfront is recognized as the striking difference between the agile practices for large-scale projects and agile principles.

This research has several implications to theory and practice. The guidelines for tailoring agile practices proposed in this research can help software development organizations that are considering the adoption of agile development methodologies. This research emphasizes the importance of using a cautious approach to adopting lightweight methodologies, ensuring their suitability and identifying different aspects of the methodologies like XP that may not be suitable under particular circumstances. We also highlight the applicability of agile development principles to large-scale, complex projects. Typically, XP has been considered to be useful only for small and medium sized projects with small team sizes. However, we suggest that even large-scale, complex projects can benefit from adapting XP to suit to their environments.

Future research will focus on identifying theoretical underpinnings of such lightweight methodologies and provide theoretical bases for tailoring such approaches. Further case studies should be conducted to refine our findings and to identify specific characteristics of projects that should be carefully considered while tailoring development methodologies, and how those characteristics can impact development methodologies.

## 6. References

[1]Manifesto for Agile Software Development. 2001, Agile Alliance:

[2]R. Baskerville, B. Ramesh, L. Levine, J. Pries-Heje, and S. Slaughter, "Is Internet-Speed Software Development Different?," *IEEE Software*, vol. Sep, 2003.

[3]K. Beck, *Extreme Programming Explained: Embrace Change*, Boston: Addison-Wesley, 2000.

[4]K. Beck and M. Fowler, *Planning Extreme Programming*, New York, NY: Addison Wesley Longman, 2001.

[5]B. Boehm, "Get Ready for Agile Methods, with Care," *IEEE Computer*, vol. 35, no. 1, 2002, pp. 64-69.

[6]A. Cockburn and J. Highsmith, "Agile Software Development: The People Factor," *IEEE Computer*, vol. Nov, 2001.

[7]B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, vol. 31, no. 11, 1988.

[8]A. Elssamadisy, "XP On A Large Project – A Developer's View," in *Proceedings of XP/Agile Universe*, Raleigh, NC, 2001.

[9]E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[10]J. Highsmith and A. Cockburn, "Agile Software Development: The Business of Innovation," *IEEE Computer*, vol. 34, no. 9, 2001.

[11]J. Little, "Up-Front Design Versus Evolutionary Design in Denali's Persistence Layer," in *Proceedings of XP/Agile Universe*, Raleigh, NC, 2001.

[12]R.C. Martin, "eXtreme Programming Development through Dialog," *IEEE Software*, vol. 18, no. 6, 2001.

[13]O. Murru, R. Deias, and G. Mugheddu, "Assessing XP at a European Internet Company," *IEEE Software*, vol. 20, no. 3, 2003.

[14]M.C. Paulk, "Extreme Programming from a CMM Perspective," *IEEE Software*, vol. 18, no. 6, 2001.

[15]J. Rasmusson, "Introducing XP into Greenfield Projects: Lessons Learned," *IEEE Software*, vol. 20, no. 3, 2003.

[16]D.J. Reifer, "XP and the CMM," *IEEE Software*, vol. 20, no. 3, 2003.

[17]B. Rumpe and A. Schr der, "Quantitative Survey on Extreme Programming Project," in *Proceedings of XP2002*, 2002.

[18]G. Schalliol, "Challenges for Analysts on a Large XP Project," in *Proceedings of XP/Agile Universe*, Raleigh, NC, 2001.

[19]C. Taber and M. Fowler, "An Iteration in the Life of an XP Project," *Cutter IT Journal*, vol. 13, no. 11, 2000.