Supporting the combination of agile manner and architecture based development method

Hongyan Yao[1, a]

[1]University of Science and Technology Liaoning, China

[a]abroat@163.com

Keywords: Software architecture, Agile, Pattern, Refactor, Architecture style

Abstract. Architecture based development methodology has its strength in organizing requirements as well as has the limitation of slowly responding to the requirements changing; while Agile manner is good at rapidly implementing components and building test scripts, but it is incapable of getting a comprehensive requirement frame. To our knowledge, there are short of related research evidences for the success combination of these two methods. In the paper we propose an approach for their coalition with one purpose: to leverage both strengths existing in these two methods for quickly responding to the changed requirements as well as speeding up the realization progress. The proposal in the paper specifies the combination steps and the communication way; meanwhile, the proposal points out the software frame should be the rendezvous point for these two methods. By a case study and the comparison to previous work the advantage of our proposal could be proven.

Introduction

Architecture-Based Software Development (ABSD) methodology is a branch of traditional way for designing and developing software. The Software Architecture (SA) design is the core work in ABSD. ABSD is suitable for the big project where a higher level framework is needed. But like Adam Solinski and Kai Petersen in [1] said: SA is a good choice in certain domain where requirements are concise, stable, and seldom changed. However, in real world, there are almost no SA plans which are designed without revision. This indicates the designed blueprint of SA needs to react to the changed requirements time to time. Unfortunately, SA is more like a static plan than a process of planning. SA is not good at rapidly responding to the requirements changing, which, as a result, will hinder the progress of ABSD. To make up the deficiency existing in ABSD, agile manner is an alternate.

Agile programming manner is firstly outlined in 2001[6, 12] and as a light-weight and flexible approach it has been applied more than a decade in industries. Critics had mentioned some of its limitations, but in this paper we take only this as its main weakness: agile manner is not good at getting requirements framework in the scope of large-scale distributed project, i.e. agile manner is incapable of building a global architecture. This weakness has been evaluated in [3] and the authors pointed out: if there is no architecture, agile may be a long-term complex process; furthermore, it's difficult to evaluate the benefits achieved. So far, commercial adoption of the agile outstrips the ability of researchers to keep up, and the related research evidence for the success of agile application remains slim.

As ABSD and agile have their own strengths and limitations, their combination may be a better solution. Zoya Durdik[8] has made a primitive discussion at this aspect and proposed a process to address the agile's limitation by introducing SA modeling. Their proposal mainly concerns about rapidly building SA in an iterative manner. To design and elicit the right requirements are their purpose. Their proposal is a suggestive process to build SA rather than a framework that leverages each methodology's strength to handle each stage located within the software engineering.

Our work in this paper focuses on the combination of ABSD and agile. After a related analysis about ABSD and agile, an approach that merges ABSD and agile manner is proposed. The framework aims to eliminate the limitations belonged to ABSD and agile, respectively. A case study and a comparison with Zoya Durdik's work will be made in the paper.

The contribution of our framework is this: 1. It defines a fine-grained process for leveraging both ABSD and agile; 2. It points out at which level the coalition of ABSD and agile is most effective; 3. It is more applicable than Zoya Durdik's.

The other parts of this paper are organized as following: in section 2 a related work is discussed; in section 3 the proposed coalition framework of ABSD and agile is specified; in section 4 a case study is presented; section 5 makes a comparison with Zoya Durdik's work; section 6 concludes the paper.

Related Work

About the agile Adam Solinski and Kai Petersen in [1] discussed what are the most significant/insignificant benefits or limitations of agile practical usage. They find companies are inclined to switch to more flexible, adjustable approaches, but the usage of agile such as pair-programming and continuous integration with testing are commonly discarded. Their work reveals the reason for that. Ahmed Sidky, et al. propose an agile adoption framework[2], which includes the index of 5 levels agility populated with agile practice and 4-stage process. It suggests the basic principles for adopting agile method at different agility level. Korhonen K's work indicates that in a large distributed organization using agile from the start may be not a good choice, and a guided SA is more valuable than coding client's story [3]. Lise Tordrup Heeager and Jeremy Rose claim that in the past decade some agile method had sprung out and were acceptable for development projects as it is lightweight and flexible, but research evidence for the success of these approaches still remains slim [4]. Particularly, Bjarnason E, et al. argue that how to get requirements in accordance with agile method is important but unsolved [5]. Their paper described five rules used to deal with this issue, but to date the influence of this manner is not evaluated. Kantorowitz E also focuses the same issue and suggests a method for requirements elicitation [7], which is different from the traditional way of UML diagrams. Still, their work needs to be evaluated.

About software architecture we have made some work in advance[9, 10]. A more completed knowledge about software architecture please refers to [11]. Here we do not discuss software architecture more than is needed, but literature [8] is worth having an attention because it shows a primitive solution for building software architecture in agile scope. Though the idea of his is coarse-grained and does not focus on the coalition of software architecture and agile, it still sheds the light on this aspect. Our work, on one hand, is related to his; on the other hand, we expand his model to a great extent and change some inappropriate arrangements.

A Proposed Frame for Coalition of ABSD and Agile

See Fig. 1. The proposed frame is built on two beliefs of ours: 1. Architecture-based methodology is a top-down method for developing software while agile-based method is a bottom-up manner. 2. No matter which method is taken, class design is the foundation. On the left of Fig. 1 there are two black long arrows. One denotes the direction of ABSD, Top-Down; the other denotes the direction of agile, Bottom-Up. These two arrows will be united at the middle layer of the frame: to confirm the changed requirements are met and current SA is relatively stable. Frame of Fig. 1 contains three layers from top to bottom: requirement layer, SA layer, and agile practice layer. The functions for each layer are specified below.

Requirement Layer. It is impossible for client to tell complete requirements in times of uncertainty, so this layer is prepared for requirements changing. In general, a changed requirement contains three aspects: changed business requirement, changed function requirement, and changed user requirement. Changed business requirement is usually taking place when certain business rule is needed to be revised or additionally added; changed function requirement means some functions are needed to be modified or added for the realization of certain business rule; changed user requirements tell that the user interface is needed to be improved to satisfy a user preference. If all requirements are updated for now, a relatively stable requirements artifact is achieved. Once requirements are changed over, the adjacent lower layer will start and fulfill it.

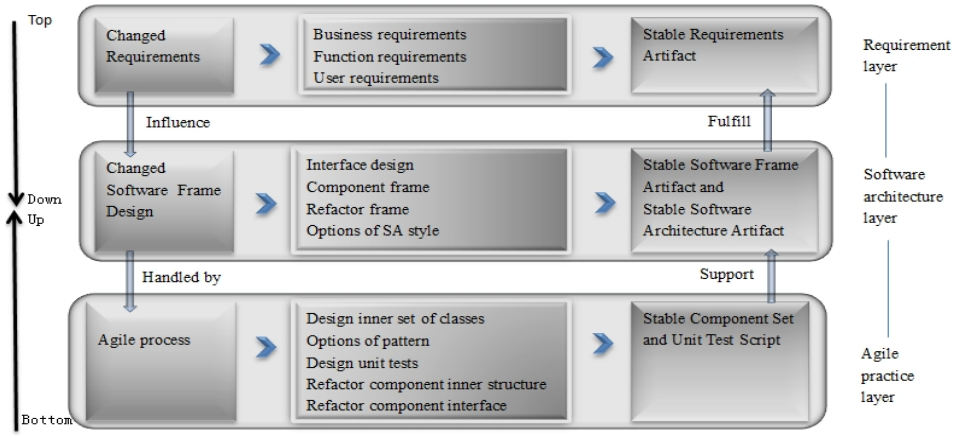

Fig 1. The coalition frame between ABSD and agile

Software Architecture Layer. The changed requirements will directly influence the building of software frame. We believe that software frame is a little different from software architecture. You could find the comparison result about architecture, structure, and framework in our previous work[11]. We emphasize only two points about frame and architecture: 1. Frame defines components and their duties while architecture describes component communication manner, i.e. using interfaces to declare communication parameters, also known as architecture style; 2. Building a runnable frame should be done at first just before the extraction of architecture, because the architecture has little sense if main components has not been defined in advance in the frame.

As influenced by the changed requirements coming from upper layer, this layer starts to change software frame. Explicitly, four stages are included:

1. Interface design. According to an important principle of behavior precedence, requirements should be defined as function signatures and classified into certain class at first so as to form primitive interfaces at class level.

2. Component frame. Sum up related interface into certain component and define component's duty are the main work of this stage.

3. Refactor frame. Recheck the interface definition and classification to find out if they are appropriate. Object sequence-interactive diagram, object collaboration diagram, or process diagram are useful in this stage. If find interface or classification is inappropriate, adjust and rebuild the component frame. We don't recommend using pattern to optimize frame at this time because the frame is perhaps not stable at this time and may be needed to adjust in future.

4. Options of software architecture style. Some software architecture styles such as pipe style, event-based style, hierarchy style, black-board style, C/S or B/S style are all the options. A primitive consideration of architecture style is necessary because it helps organize function-related component into certain layer. Also, style option guides the communication manner among components located in certain layer. After the four stages a relatively stable software frame and architecture would be the result, which are the basis for doing next: entering agile practice layer.

Agile Practice Layer. We believe it is not efficient adopting agile to get requirements at the start of a project. An isolated process for getting requirements is necessary. The practice of agile would be better if it takes the component frame as its entry point because agile is good at designing classes with respect to the interface. So, naturally, agile practice layer should base on the upper layer's software frame. Especially, the concrete component is its concern. This layer contains five stages:

1. Design inner set of classes. Within the scope of component, a set of implementing classes and auxiliary classes will be developed and encapsulated for supporting its interface.

2. Options of pattern. Sometimes there are some self-defined types that act as the parameters appearing in the function signature, and those parameters may take the subclasses' instance as input or output. Usually this phenomenon is called dependency-inversion, which is an important principle for

using pattern to solve problem. In other words, if there is an interface as a parameter appearing in function signature, pattern adoption is inevitable.

3. Design unit test. This is the character of agile: building test function time to time to check if the classes' interface is correct and works normally.

4. Refactor component's inner structure. If certain outer interface is supplemented, the component's inner structure is needed to be refactored as a despondence, like using 'decorate' pattern for adding extra handler.

5. Refactor component interface. As the interface is merely in charge of declaration, the total set of function signatures belonging to all interfaces could be arranged in any way. So if the agile process is going to re-classify some signatures, surely a suggestive interface repartition is welcome.

After these five stages, a relatively stable component set and a set of test scripts are the result. Especially, the test scripts for interface are most valuable because it will directly be used in the view boundary.

Communications between Layers. The communications between these three layers are straightforward. The changed requirements, on one hand, are gradually to form a stable requirements version; on the other hand, influences the next layer to change its former software frame design. In the same way, software architecture layer starts to change the frame so as to shape a stable frame and an architecture towards the right direction; at the same time the changed frame will be passed down to agile practice layer for handling. After handled by agile process a stable component set and unit test scripts come out, which is ready to support upper layer's stable frame; successively, the stable frame fulfills the upper layer's stable requirements artifact. Until now, one round of total iteration for meeting the changed requirements is finished.

Case Study

In this section we use an example project to demonstrate our proposal. On one hand, to prove the proposed frame is reasonable; the other hand, to verify our approach is feasible. The project is a game and it is not so large but is enough to show our idea. The game tells that an adventurer who enters a dungeon and fights his enemies, like Bat or Ghost. In the battle he encountered, he is capable to move to a new position, attack with certain weapon, or use some potion when necessary. The game contains many rounds and our hero must beat all enemies before he gets the final victory.

Initial design. As far as the above project background is concerned, a scratched initial requirements specification and a system frame design, which conform to our proposal, are shown in Table 1 and Fig. 2.

Table 1. Requirements analysis in relation to the project background

Requirements Layer	Specification
Business Requirements	Develop a round-based adventurer game.
Function Requirements	Adventurer could move, attack, pick up weapon, use weapon, etc.
User Requirements	User clicks button to control adventurer…

The Table 1 presents a specified requirements layer as a response to our proposal. Based on this temporarily stable requirements artifact, Fig. 2 shows the drawing of a temporarily stable component framework. In this frame, we simply plan to build a Game_Model for encapsulating and testing the game logic. When Game_Model is ready, we could attach a user view in any time. According to our initial design there are three roles in total appearing at the user view: Player, Weapon, and Enemy, respectively. So you could notice that in Fig. 2 we design three interfaces to support these three roles. That means the component of View can get enough data from these interfaces to paint the foreground images. These three interfaces provided by Game_Model are actually implemented by other three components; see the dependency arrow in Fig. 2. Game_Model here is more like a shell.

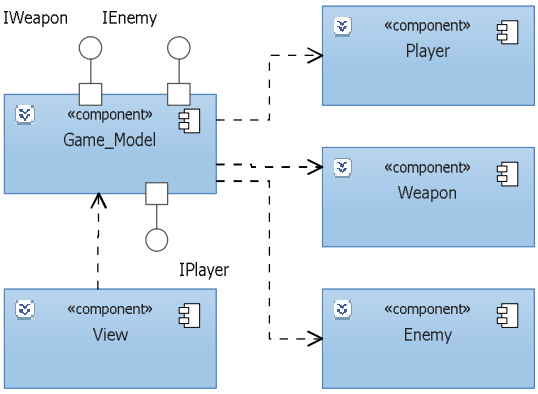

Fig 2. A corresponding framework for current requirements

According to this initial design we could suppose the MVC is a style option and Event-Handler is the main technique adopted when a graphic view is attached to Model.

Following our proposal the interfaces should be expanded in order to produce necessary data required by View Component. Fig. 3 presents a detailed interface design.

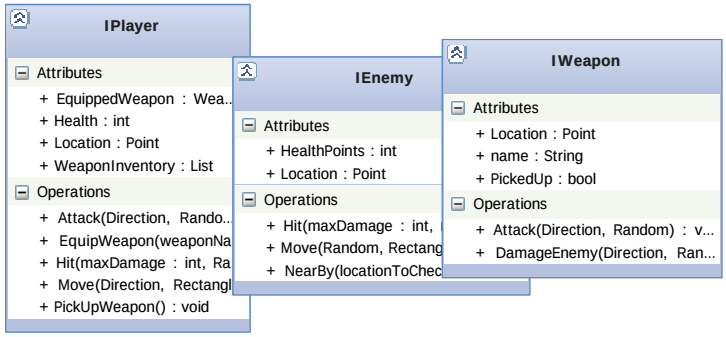

Fig 3. A temporarily stable interface design

Based on Fig. 3 we could preliminarily build test script like Fig. 4 to verify if interfaces are appropriate.

```
[TestGame_Model]
void UpdateCharacters()
{
    // init a game object for getting interfaces.
    Game aGame=new Game( ...);
    // refer to corresponding object instance to get
    // data for updating foreground images
    IPlayer aPlayer = aGame.Player;
    IEnemy [] enemies = aGame.enemies;
    IWeapon [] weapons = aGame.weaponList;
    // ... update each enemy's location and the other data ...
    foreach (var item in enemies)
        {
       ...
      // invoke IEnemy interface to get Location data
      temp_location = item.Location;// Location temp_location
        }
     // invoke IPlayer interface to get Health data
         temp_int = aPlayer.Health;  // int temp_int
    ...
}
```

Fig 4. An example of test script for checking if interfaces are declared completely

If [TestGame_Model] could pass, that means our framework runs well at interface level and the planned scenario can be simulated. Of course, most functions within interfaces have no function body, and that's just why we need to employ the Agile.

As the Agile could rapidly respond to the component inner design, i.e. Agile is excel in using the principles such as pattern, abstract, refactor, etc. to design the classes or implement function body, so as to embody the support for the interface. Fig. 5 shows a concrete classes design for the Component of

Enemy. In Fig. 5, there are abstract classes like Mover and Enemy, which are the foundation if the Component of Enemy needs to meet an extra enemy's behavior in future. Agile needs the abstract classes to embody the usefulness of Pattern. The goal of Fig. 5 is only this: to support IEnemy. The other two components also do the same thing to form a concrete design to support their interface. We do not say more than is needed here.

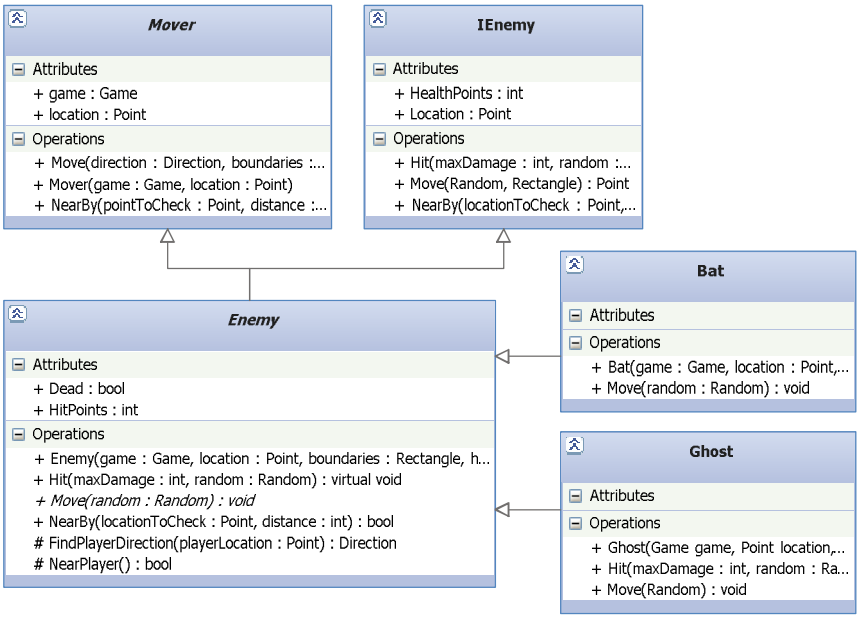

Fig 5. A set of classes designed for supporting the interface of IEnemy

After Fig. 5 the three interfaces we designed before should have their implementation and can work well. Fig. 4 also can run successfully and produce the significant and meaningful data. At this time, Game_Model is ready for any user view, and [TestGame_Model] can be invoked in any Event_Handler function.

React to a changed requirement. There is a new requirement coming after the game runs for a while. We need to save or load related data in order to avoid that we restart the game from the beginning, i.e. Game Level 1. We want to load the data we saved before and continue to play. For meeting this new requirement, according to our proposal in Fig. 1, following steps must be realized.

1. Append these Functional requirements to Table 1. To show explicitly that adventurer can save data and load data.
2. Analyze the new requirements to decide which component should be their host. Apparently, Save/Load is related to all necessary game data. Putting them into Game_Model is reasonable.
3. Add a new interface, IGame, to Game_Model Component, and then design its inner Save/Load function signatures.
4. Edit a new test script to check if new added functions are workable. (At this time Save/Load has a blank of function body)
5. Design inner class structure in Game_Model to support and implement IGame. Pattern should be considered here. The "FAÇADE" pattern is what we adopted.
6. If Save/Load functions have been implemented. Extract their function signatures and check if they are uniform to the signatures defined in step 3. If they are not, refactor the IGame of step 3 to make it uniform.

Fig. 6 shows the production of IGame. As the operation of Save or Load is a global event, it is appropriate to put them into Game_Model. In addition, we refactored the IGame to inherit from IPlayer, IWeapon, and IEnemy, for better supporting the communication between Game_Model and user view. Based on these considerations IGame has its declarations in Fig. 6, and class of Game supports it. The 5 and 6 step will omit here as the agile process for them has been specified, alike to Fig. 5.

Analysis and comparison to previous work

The proposed frame is actually an iteration process which combines top half of ABSD and bottom half of agile method. Two methods join together on software frame layer: this kind of combination will remove limitations of slowly responding to the changed requirements that exist in ABSD; at the same time it eliminates agile method's weakness of incapable planning for a global frame. Our proposal embodies the strength of rapidly building software frame and the ability of faster implementation. A comparison to [8] is illustrated in Table 2, and the explanation for the comparison is summarized below.

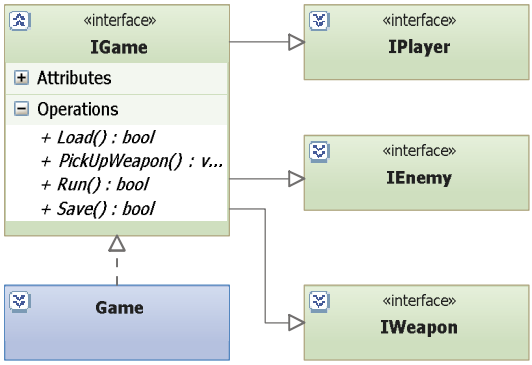

Fig 6. Design the IGame to meet new requirements

1. We don't believe that selecting pattern prior to component definition in [8] is rational for building architecture. In fact, pattern can't live without interface, because we can't evaluate the pattern whether it is appropriate when there are no interface declarations for comment. The opinions of ours in this paper show that it's better if we design the software frame prior to choosing pattern in current architecture.

2. Theirs tries to propose an agile architecture-driven model instead of a combination framework between ABSD and agile. Moreover, we can't evaluate in their proposal where the agile process should enter in and what jobs the agile process should do. Their proposal is relatively coarse-grained and does not point out specifically the usefulness of designing interface and refactoring.

3. Both theirs and ours are considering architecture design important and both proposals are in iterative manner.

Table 2. Comparison to agile architecture-driven model

The proposal	Support architecture and agile combination	Pattern selection prior to interface definition	Architecture important and Iteration necessary	Fine-grained description on interface & pattern	Considering refactor target	Support for software frame building
Agile architecture-driven model	NO	YES	YES	NO	NO	NO
Coalition of architecture based method and agile	YES	NO	YES	YES	YES	YES

Conclusion

ABSD has the limitation of slowly responding to requirements changing; while agile manner is weak in the ability of getting comprehensive requirement frame. A framework for eliminating those deficiencies is proposed. The framework contains three layers which are corresponding to changing requirements, designing revised software frame, and building runnable agile scripts. The proposal leverages both the strengths existing in ABSD and agile and aims to provide the ability of quickly responding the requirements changing and speeding up the realization progress. By analyzing a case study and making comparison with previous work, our proposal is fine-grained, feasible, and more practical. Following our proposal how to formalize and trace the changes from requirements to design will be explored in our coming paper.

References

[1] Adam Solinski, Kai Petersen, Prioritizing agile benefits and limitations in relation to practice usage, *Software Quality*, Springer, DOI 10.1007/s11219-014-9253-3, September 2014.

[2] Ahmed Sidky, James Arthur, Shawn Bohner, A disciplined approach to adopting agile practices: the agile adoption framework, *Innovations Syst Softw Eng* no.3, pp.203–216, 2007.

[3] Korhonen K, Evaluating the impact of an agile transformation: a longitudinal case study in a distributed context, *Software Quality Journal*, vol.21, no.4, pp.599-624, 2013.

[4] Lise Tordrup Heeager, Jeremy Rose, Optimizing agile development practices for the maintenance operation: nine heuristics, *Empir Software Eng*, Springer, DOI 0.1007/s10664-014-9335-7, October 2014

[5] Bjarnason E, Wnuk K, Regnell B, A case study on benefits and side-effects of agile practices in large-scale requirements engineering, *Agile RE'11*, July 26**,** 2011, Lancaster, UK.

[6] Leithiser R, Hamilton W, Agile versus CMMI - process template selection and integration with microsoft team foundation server, *ACM-SE'08*, March 28-29, 2008, Auburn, AL, USA.

[7] Kantorowitz E, Verbal use case specifications for informal requirements elicitation, *Lecture Notes in Computer Science*, 2014:165-174. DOI 10.1007/978-3-642-54894-9_12

[8] Zoya Durdik, Zoya Durdik, Towards a process for architectural modeling in agile software development, *Proceedings of the joint ACM SIGSOFT conference -- QoSA and architecting critical systems -- ISARCS*. ACM, 2011:183-192.

[9] Hongyan Yao and Yunji Ma, An Exploration for the Software Architecture Description Language of WRIGHT, *ICIC-EL*, vol.8, no.12, pp. 3481-3487, December 2014.

[10] Hongyan Yao,Yefeng Jiang, Wenxuan Shen, A Revised Orthogonal Software Architecture COA, *ICIC-EL*, vol.7, no. 8, pp. 2255-2261, July 2013.

[11] Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice (3rd Edition)*, Addison-Wesley Professional, ISBN: 978-7-302-31293-2, October, 2012

[12] Robert C. Martin, Micah Martin, *Agile Principles, Patterns, and Practices in C#*, Prentice Hall, ISBN: 0131857258, July 2006.