2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Towards an Architectural Framework for Agile Software Development

Richard Mordinyi, Eva Kühn
Space-based Computing Group
Vienna University of Technology
1040 Vienna, Austria
{*rm,ek*}*@complang.tuwien.ac.at*

Alexander Schatten
Complex Systems Desing & Engineering Lab
Vienna University of Technology
1040 Vienna, Austria
alexander.schatten@tuwien.ac.at

***Abstract*—One of the ideas of agile software development is to respond to changes rather than following a plan. Constantly changing businesses result in changing requirements, to be handled in the development process. Therefore, it is essential that the underlying software architecture is capable of managing agile business processes. However, criticism on agile software development states that it fails to pay attention to architectural and design issues and therefore is bound to engender suboptimal design-decisions.**

In this paper we propose an architectural framework for agile software development, that by explicitly separating computational, coordinational, and communicational models offers a high degree of flexibility regarding architectural and design changes introduced by agile business processes. The framework strength is facilitated by combining the characteristics and properties of state-of-the-art middleware architectural styles captured in a simple API.

The benefit of our approach is a clear architectural design with minimized effects of changes the models have on each other, accompanied by an efficient realization of new business requirements.

***Keywords*-Architectural Styles, Agile Business Requirements, Agile Software Development, Decoupling, Abstraction**

I. Introduction

Business constantly changes. Therefore, software architectures should be able to manage agile business processes and need to have the ability to meet future changes and business needs. The field of agile software development [1], [2] (ASD) addresses exactly the challenges of an unpredictable, turbulent business and technology environment. Thus, the question is how to better handle architectural changes, while still achieving high quality.

For distributed systems it is essential to make use of a flexible and adaptable platform that can respond to new requirements in an efficient way. Consequently, the usage of appropriate architectural styles for the design of software systems is a challenge. A common approach towards creating flexible, dynamic business processes and agile applications is the service-oriented computing style (SOA) [3]. For instance, the Enterprise Service Bus (ESB) [4] promises to interconnect and route services in a loosely coupled manner for a clear separation of business logic and integration logic. However, an ESB routes service data from one application to another and usually does not keep the history of messages and service interaction, i.e. does not maintain a global state.

Thus, the main question in software development regarding software architecture still remains [5]. On the one hand how many various numbers of eventualities have to be taken into consideration, and therefore how much time and effort should be invested into design and implementation of components with respect of a good architectural design to cover all these circumstances, which eventually at the end may be not used at all. On the other hand, no or hardly any planning ahead, and at the same time bearing the risk of redesigning the existing architectural design from the scratch, once it is not capable of handling the latest requirement. All in all, it means that architecture and business do not evolve in the same way and same ”speed” [6]. Problems regarding architectural and design issues in ASD have been discussed in several papers, like [7], [8], [9], [10], stating that ASD fails to pay attention to architectural and design issues and therefore is bound to engender suboptimal design-decisions.

In previous work [11], we argued that although, software systems are usually not built by means of a single architectural style, there is a tight coupling between the application and the used style. This implies adaptations of the application in case the middleware has to be altered due to changing business requirements.

Extending the ideas in [11], in this paper we propose the Architecture Framework for Agile processes (AFA)[1], in which it is explicitly distinguished between computational logic, coordinational and communicational models. The three models are independent of each other and therefore AFA offers a high degree of flexibility regarding architectural and design changes introduced by agile business processes. Like in [11], our approach combines and includes the characteristics and properties of the major architectural styles found in distributed middleware (section IV) captured in a simple API. AFA can be seen as an abstraction layer between applications and architectural styles, and as such it provides loosely coupling between the applications and their way of coordinating each other, between applications

[1]an implementation of the framework, called Mozartspaces, can be downloaded at www.mozartspaces.org

978-0-7695-4005-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ECBS.2010.38

and architectural styles, between the applications and the way they exchange information.

The benefits of the proposed AFA approach are a) the efficient realization of changing business requirements affecting the underlying architecture; and b) adaptations of the architecture transparent to the application resulting in less complex application logic since it can entirely focus on its business process.

The remainder of this paper is structured as the following: section II pictures the use case, section III defines research questions, section IV summarizes related work, section V describes the concept and the architecture, while section VI discusses the proposed concept. Finally section VII concludes the paper and proposes further work.

II. Scenario

In this section, we introduce a fictive scenario, based on an insurance company and its agents in field services, which should demonstrate the need for a change in the architecture due to new business requirements. In section V-C we explain how AFA abstracts these transitions.

As a starting point let's assume that agents visiting potential customers fill in insurance related forms at the customer. Due to technical and economical reasons the mobile agent needs a permanent connection to the main insurance server of the company, both physically via e.g., UMTS and logically to its services. However, the required permanent connection between the agents and the main server hinders the agents to work efficiently with their customers. The agents cannot be sure whether the transmission capabilities of the provider cover the area where the customer lives, leading to an unreliable customer information management. This brings in a new requirement demanding the agent capability of working offline as well, without being dependable on a permanent connection. However, this leads to a break in the architecture in the sense that data stored before on the main server only, has to be partially replicated to the agents' mobile devices and persisted there. Therefore, both the server and the application need to manage their own data and need to have the capability of synchronizing data changes.

III. Research Questions

Inspired by agile software development and based on the limitations of traditional middleware technologies with respect to introduction of new business requirements and their effects, we wanted to investigate a) the advantages and limitations of the proposed approach with respect to changing business requirements, b) what are the advantages of decoupling the three models, and c) how to realize changing business requirements transparent to the participating clients.

IV. Related Work

This section summarizes related work on architectural styles and agile software development.

A. Architectural Styles

Distributed middleware are mostly based on either dataflow style, such as pipes-and-filters, on data-centered style, i.e. a repository, or on implicit invocations, like publish-subscribe or event-based [12].

1) Dataflow Architectural Style: Pipes-and-filters, representing the dataflow style, define independent components (filters) that can be connected with each other but which do not know about the existence of other filters [13]. The connections between filters determine the pipeline. Sharing data between filters is only possible by passing it from one filter to the next, even if it is not needed in an intermediary step. SOA [3] typically makes use of the pipes-and-filters style. Services can be implemented as filters and the way of routing messages determines the pipeline that represents the business logic. The ESB [4] is the major platform used in SOA offering the necessary functionality in order to make use of SOA. The ESB discards any service-relevant data after message delivery. Thus, it cannot offer a shared repository that clients can use in order to coordinate themselves.

2) Data-centered Architectural Style: The essence of data-centered styles is that multiple components have access to the same data store, and communicate through that data store. A shared repository does provide its clients with access to shared data. Databases are the typical representation of this data-centered architectural style. Active repositories tie together the shared repository with another architectural style, which are event-based systems [14]. An active repository is able to notify registered clients about changes [13]. A repository does not provide the means for specifying in which order its shared data needs to be processed by its clients. Thus, repositories cannot offer routing capabilities in order to determine the processing sequence among its clients. Thus, it is irrelevant for the usage in pipes-and-filters.

Another data-centered architectural style is the blackboard based one, in which the state of the information on the blackboard determines the order of execution. A representative of the style is e.g., the Linda coordination model by David Gelernter [15]. It describes the usage of a logically shared memory, called tuple space, by means of simple operations (out, in, rd, eval) as a communication mechanism for parallel and distributed processes.Unlike Linda, AFA (section V) allows e.g., storing shared data in a customizable structured way. This facilitates the efficient implementation of coordination concerns among middleware clients.

In [16] an extension to the pipes-and-filters style was proposed, where a shared repository is also supported. However, the hybrid framework does not offer the abstraction of the pipes-and-filters style but rather adds shared data to the pipeline.

3) Implicit Invocation Architectural Style: This style is characterized by calls that are invoked indirectly and implicitly as a response to a notification or an event. The

group is represented by the publish/subscribe [17] and event-based [14] architectural styles.

B. Agile Software Development

Concepts for agile software development (ASD) have been created by experienced practitioners and can be seen as a reaction to e.g., plan-based methods, which attach value to "a rationalized, engineering-based approach" [18]. There, problems are seen to be fully specifiable and solvable with an optimal and predictable solution. By means of extensive planning and codified processes development can be made efficient and predictable. By contrast, ASD has been proposed as a solution to problems resulting from an unpredictable world. Several agile methods have evolved over time, like XP [19] or Scrum [20]. However, there is also skepticism [2] regarding ASD with respect to architecture design and implementation issues. One of them is that agile development is an excuse for developers to implement as they like, coding away without proper planning or design [7], [5] and consequently causing suboptimal design-decisions [8], [9].

V. Architecture

This section pictures the idea of AFA in detail. It gives a brief introduction of its components and explains how decoupling between the three models is achieved. Finally, it describes how the given change in the scenario (section II) can be realized with the proposed concept.

A. The Architectural Framework for Agile Processes

The main component of the AFA architecture are Internet addressable containers [21] which is a collection of entries accessible via a basic simple API. The container's interface provides an API for reading, taking, and writing entries, but extends the original Linda API with the methods `destroy`, `shift` and `notify`. Destroy removes an entry from the container, while shift writes an entry after it has removed one. Another important component is the so called coordinator [21] which are programmable parts of the container being responsible for managing their view on the entries in the container. The aim of a coordinator is to represent a coordination model and to structure and organize the entries in the container for efficient access [22]. The difference to Linda is that a container may be bounded to a maximum number of entries, and allows the usage of so called coordinators with each having its specific and optimized view on the stored entries. In contrast to databases, AFA offers blocking operations known from the Linda model, thus allowing queries for future data states. Furthermore, databases need a static data model of the entries they have to store, while containers allow the usage of several different coordinators at the same time, enabling efficient dynamic data models, and thus being schema-free.

The last main component of AFA are the so called aspects [22], which represent additional computational logic and are executed on the peer where the container is located. Aspects are triggered by operations on a specific container, rather on the according impact. Aspects can be located before or after the execution of an operation and added and removed at any time during runtime. A detailed explanation of how aspects work and the interrelation between aspects and containers is given in [22].

B. Supported ways of Decoupling in AFA

In [11] we have described how different architectural styles can be combined to manage architecture limiters and breaker. The concept identified several layers, each responsible for different tasks. By means of combining the different responsibilities of each layer a specific architectural style with specific configuration has been created. However, those layers can be explicitly clustered and categorized (figure 1) resulting in models distinguished by their capabilities for managing computational, coordinational, or communicational requirements.

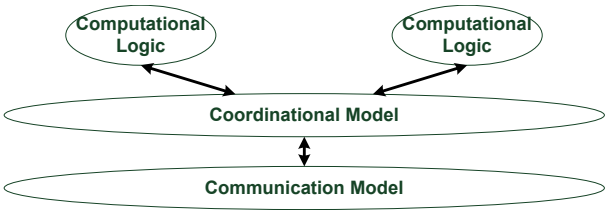
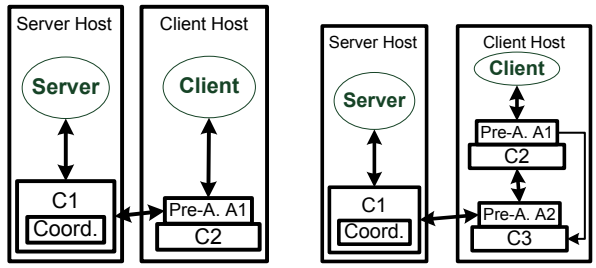


Figure 1: Aspects of Decoupling.

As stated in [15] the computation model is used to express the computational requirements of an algorithm, while the coordination model is used to express the coordinational requirements. On the one hand, this decoupling allows to change the application without having an effect on the way it coordinates itself with other applications. On the other hand, a fifo style of coordination - comparable to message-based communication - can be switched to e.g., lifo style of coordination, or more complex coordination models [21].This is done by replacing the existing coordinator in the container. In traditional sense, the computational and coordination models are combined, since a lot of the systems rely on the pipes-and-filters or call-and-return architectural styles. This implies that the application itself also has to contain and implement the complexities coming along with the used coordination model.

However, so far it has only be defined how and which applications coordinate each other. It has not been specified whether the applications run in the same process, on the same machine, or distributed on the Internet. Furthermore, it has not been specified how the necessary information needed for coordination is exchanged between the participating application of the coordination model.



(a) Before the new requirement. (b) After the new requirement.

Figure 2: Scenario 1

This part is done by the communication model. To be correct, this model needs to be divided into a dissemination/distribution model and a the real communication model. Since containers store entries, the dissemination model specifies where the container and its data have to be stored physically, whether there are multiple replica of it and how they are kept consistent [22]. The communication model describes how data from one container is transmitted to the other. The model may contain lower level protocols like udp, tcp, or higher level ones like P2P protocols.

A big contribution to this picture is done by the aspects. As mentioned aspects may contain any computational logic, and may return several different values [22] once that logic has been executed. In combination this adds extra value to the framework. As an example, a container may host a pre-aspect that does nothing else, but rerouting incoming operations to other services and then skipping the operation. This implies that the container will never be filled with data. However, the application has not been changed, it still uses the same coordination model, but data is not retrieved from or written into the container. As an example the data may be retrieved from and written to a database via a WebService call.

C. Handling Agile Business Requirements with AFA

This section intends to explain the idea of AFA by means of the scenarios described in section II. The new business requirement in the first scenario is to allow the agents to work offline. Figure 2a shows the used components in an AFA environment before the new requirement. The figure shows that both, server and client part, have a container running on their machines. However, only the server part keeps the data accessed by the clients. Clients access the data by accessing the local container C2. However C2 does not contain any data, since pre-aspect A1 intercepts the operation and reroutes it to the server.

The new requirements state that some data needs to be stored at the client as well. A very simple solution to that problem is shown in figure 2b. Instead of forwarding operations to container C1, pre-aspect A1 now makes a copy of it and executes the operation on container C3 as well. This container is needed to keep track of changes while the client is offline in order to be capable of executing a synchronization strategy once the client is online again. Synchronization is done by pre-aspect A2, that checks connectivity status and updates container C2, once the synchronization process has started.

VI. Discussion

In this paper, we propose the concept of an Architecture Framework for Agile processes (AFA) in order to allow the realization of new business requirements. The advantages of the proposed approach with respect to changing business requirements are that the applications do not have to consider a) the underlying architectural style, b) the used coordination model, c) the used communication technology, or d) the way how data is disseminated. However, an abstraction technology placed between middleware and client application causes an additional overhead which affects overall performance. Nevertheless, the proposed concept allows decreasing development and migration time by reducing the effort needed to adapt the current system to new business requirements and therefore it saves costs while improving adaptability and re-usability.

Summing up, we take a look at the quality attributes defined in [23].

Evaluations have to be performed, but we think that the overall `performance` decreases due to the additional layers in AFA.

`Security` is supported transparently to client applications by adding security-relevant aspects to the AFA middleware.

As shown in [22] containers in AFA can be transparently replicated in order to improve availability and fault tolerance.

With respect to `usability`, AFA offers a generic interface. Thus, components always act upon the same interface, which improves `modifiability`, modularization and encapsulation. By abstracting the underlying architectural style AFA can be ported to many different middleware technologies, such as inter-process communication (e.g. RMI), SOA, etc.

A specific constellation of containers, their coordinators, and installed aspects can be seen as a pattern for a specific problem. In this sense AFA facilitates `re-usability`.

Since changes are restricted to one of the three models, other components can be tested in the same manner which improves `testability`.

VII. Conclusion and Future Work

In this paper, we described the concept of the Architecture Framework for Agile processes (AFA) as an abstraction framework in order to allow the efficient realization of new business requirements with minimal effects on other components in the architecture.

Based on the components AFA provides, and the explicit separation between computational, coordinational, and communicational model a high degree of flexibility has been achieved with minimal effects on each other in case one of the models has to be adapted due to new business requirements.

The benefits of the approach are a rigid decoupling between the models and architectural styles allowing changes in the architecture with minimal effect on other components, resulting in less testing and therefore minimized time-to-market. Since new requirements can be mapped to one of the models, the time needed to adapt the system reduces, and therefore saves costs.

Further work will include a benchmarking of the framework, i.e. to what extent does the additional abstraction layer decrease computational performance. A more comprehensive evaluation with respect to testing and development time is intended. The latter case will also investigate the influence of software development experience of software engineers on software development with AFA support.

REFERENCES

[1] J. Highsmith and A. Cockburn, "Agile software development: the business of innovation," *Computer*, vol. 34, no. 9, pp. 120–127, Sep 2001.

[2] T. Dingsoyr and T. Dyba, "What do we know about agile software development?" *Software, IEEE*, vol. 26, no. 5, pp. 6–9, Sept.-Oct. 2009.

[3] M. P. Papazoglou and W.-J. Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, vol. 16, no. 3, pp. 389–415, 2007.

[4] D. Chappell, *Enterprise Service Bus*. O'Reilly Media, Inc., 2004.

[5] E. Hadar and G. M. Silberman, "Agile architecture methodology: long term strategy interleaved with short term tactics," in *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. New York, NY, USA: ACM, 2008, pp. 641–652.

[6] M. Pikkarainen and U. Passoja, "An approach for assessing suitability of agile solutions: A case study," *Extreme Programming and Agile Processes in Software Engineering*, pp. 171–179, 2005. [Online]. Available: http://dx.doi.org/10.1007/11499053_20

[7] S. R. Rakitin, "Manifesto elicits cynicism," *IEEE Computer*, vol. 34 (4), 2001.

[8] P. McBreen, *Questioning Extreme Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002, foreword By-Beck, Kent.

[9] M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case Against XP*. Apress, Berkeley, 2003.

[10] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Inf. Softw. Technol.*, vol. 50, no. 9-10, pp. 833–859, 2008.

[11] R. Mordinyi, E. Kühn, and A. Schatten, "Space-based architectures as abstraction layer for distributed business applications," in *Accepted for the Track on Software Engineering for Distributed Systems at the 4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS-2010) (TechRep at http://tinyurl.com/ylnosxb)*, 2010.

[12] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[13] P. Avgeriou and U. Zdun, "Architectural patterns revisited - a pattern language," in *Proc. Of 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, 2005.

[14] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[15] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, 1992.

[16] A. R. Franois, "Software architecture for computer vision: Beyond pipes and filters," Technical Report IRIS-03-240, Institute for Robotics and Intelligent Systems, USC, Tech. Rep., 2003.

[17] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[18] S. Nerur, R. Mahapatra, and G. Mangalaraj, "Challenges of migrating to agile methodologies," *Commun. ACM*, vol. 48, no. 5, pp. 72–78, 2005.

[19] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[20] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[21] E. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber, "Introducing the concept of customizable structured spaces for agent coordination in the production automation domain," in *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 625–632.

[22] E. Kühn, R. Mordinyi, L. Keszthelyi, C. Schreiber, S. Bessler, and S. Tomic, "Aspect-oriented space containers for efficient publish/subscribe scenarios in intelligent transportation systems," *The 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA'09)*, 2009.

[23] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.