# Structuring Complexity Issues for Efficient Realization of Agile Business Requirements in Distributed Environments

Richard Mordinyi, Eva Kühn, and Alexander Schatten

Space-based Computing Group
and
Christian Doppler Laboratory "Software Engineering Integration for Flexible Automation Systems"
Vienna University of Technology
1040 Vienna, Austria
{richard.mordinyi,eva.kuehn,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 lacks paying attention to architectural and design issues and therefore is bound to engender suboptimal design-decisions. We propose an architectural framework, that by explicitly distinguishing computational, coordinational, organizational, distributional, and communicational models offers a high degree of flexibility regarding architectural and design changes. The framework strength is facilitated by a) combining the characteristics and properties of architectural styles captured in a simple API, and b) offering a predefined architectural structure to the developer of distributed applications to cope with complexities of distributed environments. The benefit of our approach is a clear architectural design with minimal mutual effects of the models with respect to changes, accompanied by an efficient realization of new business requirements.

**Keywords:** Agile Business Requirements, Agile Software Development.

## 1 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] addresses exactly the challenges of an unpredictable, turbulent business and technology environment. Thus, the main question [2] in software development regarding software architecture is how to handle these changes better, while still achieving high quality. On the one hand, how many various numbers of eventualities have

A. Sillitti et al. (Eds.): XP 2010, LNBIP 48, pp. 202–207, 2010.

to be taken into consideration, and therefore how much time and effort should be invested into design and implementation of components or layers with respect to a good architectural design to cover all these circumstances, which eventually at the end may be not used at all. And on the other hand, performing 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 [3]. The former case means, that software developers a) do not really focus on realizing the current requirement, but b) develop components which might not be needed in future. This results in higher development time and costs. Problems regarding architectural and design issues in ASD have been discussed in several papers, like [4], [5], [6]stating that ASD lacks paying attention to architectural and design issues and therefore is bound to engender suboptimal design-decisions.

In this paper we propose the Architecture Framework for Agile business requirements (AFA)[1] in ASD, in which it is explicitly distinguished between computational logic, coordinational, organizational, distributional, and communicational models. The five categories are independent of each other and therefore AFA offers a high degree of flexibility regarding architectural and design changes introduced by agile business processes. In the proposed framework the categories are not implemented by means of the software layer pattern [7] but rather in a component-oriented style [8], while making usage of the interceptor pattern. This allows to keep changes local or extend categories individually. 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 and architectural styles, between the applications and the way they process, distribute, and exchange information.

## 2 Related Work

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 [9]. A detailed description regarding the advantages and limitations of the combination of architectural styles can be found in [10].

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" [11]. By contrast, ASD has been proposed as a solution to problems resulting from an unpredictable world, and several agile methods have evolved over time, like Dynamic Systems Development Method [12], or XP [13].However, there is also skepticism [14] 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 [4], [2] and consequently causing suboptimal design-decisions [5], [6].

[1] An implementation of the framework can be downloaded at `www.mozartspaces.org`

Software architectures for distributed systems are a challenge in terms of software development and evolution. Yet it is known from experience that evolution is a key problem in software engineering and exacts huge costs, thus companies spend more resources on maintenance (i.e. evolving their software) than on initial development [3]. Experiences [15], [16] gained by several case studies revealed most influential risks and their affects on the design of the software architecture. The range of affects starts at a complete refactoring of the architecture e.g., in case of poor clustering and ends at adding architectural add-ons.

## 3 Architecture of the AFA Framework

The main components [17] of the AFA Architecture are Containers, Coordinators, and Aspects. A container is a collection of entries accessible via a basic simple synchronous and asynchronous API of read, take, write, and destroy operations. In case of read, take, and destroy, the operation may be blocked in case the queried entry is not available. Coordinators are programmable parts of the container and are responsible for managing their own view on the entries in the container. A container may host multiple coordinators which aim is to represent a coordination model and to structure and organize the entries in the container for efficient access. Finally, a container may be surrounded by a set of pre- and post-aspects, representing additional computational logic and deployed at specific interception points according to the interceptor pattern. They are executed on the peer where the container is located and are capable of changing or to extending the operation with further functionalities, e.g., to replicate container entries [18].

The three main concepts of the framework form a predefined structure for the developer of distributed applications to cope with complexity issues of distributed systems. These issues can be explicitly clustered and categorized resulting in models distinguished by their capabilities for managing computational, coordinational, organizational, distributional, and communicational requirements.

The computational category represents the application logic and thus the business requirement. It specifies the behavior of the application in case of receiving data and when to write new data into which containers. For the application a container may look like an endpoint known from ESBs [19]. However, an endpoint abstracts underlying protocols trying to map high-level process flows into individual service invocations. A container abstracts transportation protocols as well, but offers the capability to use other coordination models beside the FIFO coordination style representing simple messaging.

The coordination model is used to express the communication and synchronization requirements between applications. This decoupling allows to switch between coordination models [17] transparent to the application accessing the container 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. Furthermore, from the point of view of the application it is "'just"' reading and writing entries into a container. However, although the container has an entry - written by another application - it may still not be accessible since from the point of view of the coordination model the 'coordination strategy' may not be yet complete.

The organizational category hides from the application the various combinations of architectural styles [10], offers space for managing cross-cutting concerns, and abstracts resources. Architectural styles explicitly designed for coordination issues, like the blackboard style (see section 2), have already been taken into consideration in the coordinational category. A big contribution to this category is given by the aspects as they may contain any computational logic a) restricted to the aspect only, like filter functionalities, or b) logic that needs access to other resources, like in the case of aggregating events or logging operations.

The distributional category deals with transparency issues known from distributed systems, like location-, migration-, relocation-, and replication-transparency. From the application point of view it is not known whether the container is embedded, hosted locally, or on a single server or in a P2P network. Similar to an ESB, AFA enables to migrate a single container executed on a server to a replicated one. However, by means of aspects the container can be located and replicated in a P2P environment. In such case, aspects installed at every replica would define the replication technique and the consistency strategy between the replicas [18]. Since aspects may contain any code, for software developers it is not necessary to implement replication issues from the scratch. Aspects may contain already existing solutions to these issues, like realized in group communication protocols.

The communicational category allows to specify how the necessary information needed for coordination is exchanged between the participating containers. It describes how data from one container is transmitted to the other. The model may contain lower level protocols or higher level ones like P2P protocols. However, in case the developer decides to use group communication protocols, as mentioned before, AFA cannot modify the settings of that product, since it is not it the scope of AFA and thus the communicational category anymore.

## 4 Discussion

The proposed AFA framework explicitly distinguishes between five categories each responsible to manage specific complexity issues of distributed environments. This helps system engineers to identify only those components which are effected by a new business requirement, and thus minimize the effects of the implementation of the new requirement on other categories.

Todays middleware technologies represent a specific architectural style [10]. This implies that the middleware itself either does not have the ability or would require a lot of effort to satisfy business requirements the middleware has not been explicitly designed for. On the contrary, the AFA framework abstracts the combined power of several architectural styles and thus facilitates the implementation

of new requirements by specific changes in the AFA structure (coordinators and set of aspects). The advantage of such a structure is that an average engineer [20] with limited knowledge about e.g., design patterns, object-oriented programming, or available frameworks operates with simple concepts in each category. This limits the numbers of design and implementation decisions an average engineer would have to make. This also means that in case of design mistakes and implementation errors the effects of the decision is limited to the category where it has been made. The limitation of the concept is that professional engineers may be restricted in their way of implementing. Since they have the right knowledge, they know how to tune frameworks or how to design architectures in combination with the right design patterns in order to a flexible and efficient software design. The AFA structure would limit they degree of freedom with respect to design decisions.

## 5 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. The concept is capable of representing the characteristics of different architectural styles at the same time and by explicitly distinguishing between computational, coordinational, organizational, distributional, and communicational models, it offers a high degree of flexibility. The combination of a generic interface, containers, coordinators and aspects a) offers a predefined architectural structure to the developer of distributed applications to effectively cope with complexities of distributed environments, and b) allows to decouple the models resulting in minimal mutual effects in case of changes transparent to the application. 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.

## References

1. Highsmith, J., Cockburn, A.: Agile software development: the business of innovation. Computer 34, 120–127 (2001)
2. Hadar, E., Silberman, G.M.: Agile architecture methodology: long term strategy interleaved with short term tactics. In: OOPSLA Companion 2008: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 641–652. ACM, New York (2008)
3. Jazayeri, M.: On Architectural Stability and Evolution. In: Blieberger, J., Strohmeier, A. (eds.) Ada-Europe 2002. LNCS, vol. 2361, pp. 13–23. Springer, Heidelberg (2002)
4. Rakitin, S.R.: Manifesto Elicits Cynicism. IEEE Computer 34(4) (2001)
5. McBreen, P.: Questioning Extreme Programming. Addison-Wesley Longman Publishing Co., Inc. (2002)

6. Stephens, M., Rosenberg, D.: Extreme Programming Refactored: The Case Against XP. Apress, Berkeley (2003)
7. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture: Patterns for Resource Management. John Wiley & Sons, Chichester (2004)
8. Heineman, G.T., Councill, W.T. (eds.): Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc. (2001)
9. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations. In: Theory, and Practice. Wiley Publishing, Chichester (2009)
10. Mordinyi, R., Kühn, E., Schatten, A.: Space-based Architectures as Abstraction Layer for Distributed Business Applications. In: Track on Software Engineering for Distributed Systems at the 4th International Conference on Complex, Intelligent and Software Intensive Systems, CISIS 2010 (2010)
11. Nerur, S., Mahapatra, R., Mangalaraj, G.: Challenges of migrating to agile methodologies. Commun. ACM 48, 72–78 (2005)
12. Stapleton, J.: DSDM: Business Focused Development. Pearson Education, London (2003)
13. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Professional, Reading (2004)
14. Dingsoyr, T., Dyba, T.: What Do We Know about Agile Software Development? Software, IEEE 26, 6–9 (2009)
15. Slyngstad, O.P., Li, J., Conradi, R., Babar, M.A.: Identifying and Understanding Architectural Risks in Software Evolution: An Empirical Study. In: Jedlitschka, A., Salo, O. (eds.) PROFES 2008. LNCS, vol. 5089, pp. 400–414. Springer, Heidelberg (2008)
16. Slyngstad, O., Conradi, R., Babar, M., Clerc, V., van Vliet, H.: Risks and Risk Management in Software Architecture Evolution: An Industrial Survey. In: Software Engineering Conference, APSEC 2008. 15th Asia-Pacific, pp. 101–108 (2008)
17. Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C.: Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009, pp. 625–632 (2009)
18. Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C., Bessler, S., Tomic, S.: Aspect-oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems. In: The 11th International Symposium on Distributed Objects, Middleware, and Applications, DOA 2009 (2009)
19. Chappell, D.: Enterprise Service Bus. O'Reilly Media, Inc., Sebastopol (2004)
20. Hannay, J.E., MacLeod, C., Singer, J., Langtangen, H.P., Pfahl, D., Wilson, G.: How do scientists develop and use scientific software? In: SECSE 2009: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, pp. 1–8. IEEE Computer Society, Los Alamitos (2009)