Iterative and Incremental Development of Component-Based Software Architectures

Colin Atkinson and Oliver Hummel
Software Engineering Group
University of Mannheim
68131 Mannheim, Germany
Phone: +49 6 21 / 1 81 – 39 29
{atkinson, hummel}@informatik.uni-mannheim.de

ABSTRACT

While the notion of components has had a major positive impact on the way software architectures are conceptualized and represented, they have had relatively little impact on the processes and procedures used to develop software systems. In terms of software development processes, use case-driven iterative and incremental development has become the predominant paradigm, which at best ignores components and at worse is even antagonistic to them. However, use-case driven, I&I development (as popularized by agile methods) and component-based development have opposite strengths and weaknesses. The former's techniques for risk mitigation and prioritization greatly reduce the risks associated with software engineering, but often give rise to suboptimal architectures that emerge in a semi-ad hoc fashion over time. In contrast, the latter gives rise to robust, optimized architectures, but to date has poor process support. In principle, therefore, there is a lot to be gained by fundamentally aligning the core principles of component-based and I&I development into a single, unified development approach. In this position paper we discuss the key issues involved in attaining such a synergy and suggest some core ideas for merging the principles of component-based and I&I development.

Categories and Subject Descriptors: D.2.11 [**Software**]: Software Architectures

General Terms: Design

Keywords: Component-based software architectures, iterative and incremental software development.

1. INTRODUCTION

Software component technologies have come a long way since the idea of building new applications from prefabricated building blocks was first proposed in the 1960's [1], and if one includes services as special kinds of components, they now form the backbone of most large-scale enterprise system architectures [8].

CBSE '12, June 25-28, 2012, Bertinoro, Italy.

However, the value of components in software engineering stems almost exclusively from their static role as structural artefacts in software architectures rather than from their contribution to the processes and procedures of software engineering. The notion of components [2] has made little practical contribution to the state-of-the-art in software processes since the idea of software reuse was en vogue in the 1980's and all recent process innovations in software engineering have taken place independently of component-based architectures. At best, today's leading software engineering processes ignore components, and at worst they can be regarded as being incompatible with them. In other words, the state-of-the-art in software processes has effectively become decoupled from the state-of-the-art in software architectures.

Since the turn of the century, two main process innovations have had a significant impact on mainstream software engineering practices. One is model driven development and the other is iterative and incremental ("I&I") development. Model-driven development [10] accelerates the development processes by raising the level of abstraction at which software systems are represented and by semi-automating the process of generating executable code. In contrast, I&I developmen (today most widely applied under the banner of agile development [17]) lowers the risks and costs associated with software development by organizing development projects in terms of small mini-projects rather than a single, "big bang" waterfall project. Although they are essentially independent, these two paradigms are commonly used together in a synergetic way to combine their benefits. Well known examples that integrate I&I and modelling include the RUP and Agile Modelling [4].

I&I based development and model-driven development are both fundamentally independent of the notion of components, and neither support nor discourage their use. In effect, they are both agnostic to components. Interestingly, several methods have attempted to integrate model-driven development and component-based development principles such as KobrA [11], Catalysis [15] and UML-Components [16], although none of these has taken off in commercial software development so far. However, to the best of our knowledge no development methods have been defined which attempt to combine the advantages of component-based and I&I based development for mainstream software engineering. This is perhaps not surprising since at first sight the paradigms seem to be totally unrelated to one another. It is also unfortunate because the strengths of one paradigm are weaknesses of the other and vice versa. Component-based development is strong in terms

of the quality of the software architectures that is supports, but weak in terms of the sophistication of the processes used to generate them, while I&I development is strong in terms of the organization of process steps and activities involved in software engineering, but weak in terms of the quality of the architectures that it typically delivers. A method that combines their strengths and ameliorates their weaknesses could therefore have a big impact. In this position paper we explore the possibility of achieving this goal and show how the core concepts of I&I development could be re-incarnated in the context of component-based developed to create a new approach whose strengths are the union of their individual strengths.

In the following section we briefly discuss the current state of component technologies before we contrast them with architectural practices in today's I&I (mainly agile) approaches in section 3. Section 4 then introduces our approach for reconciling the two in order to achieve component-oriented increments that may significantly reduce architectural refactoring effort. We continue our paper with a discussion of our proposal in section 5 and conclude it with a summary of our contribution and our findings in section 6.

2. DISTINCT COMPONENT MODELS

Early so-called component technologies, such as EJBs, COM resp. COM+ and CORBA, that – from today's point of view – can be seen as essentially object-oriented middleware frameworks for distributed systems, distinguished between the process of developing components and the process of assembling them into new applications. They assumed a "flat" component architecture in which a flat (i.e. non-nested) set of fine-granular "components" is glued together to create a system, but this system is not itself regarded as a component that can form part of a larger system. In such "**flat" component models**, components are effectively little applications in their own right that can be implemented using any mainstream development technology such as Java or C#. From the perspective of the "glue code" orchestrating the interaction of the building blocks, components are black-boxes that can be brought together in certain limited ways to deliver the desired properties of the system as a whole. This view is illustrated schematically in Figure 1.

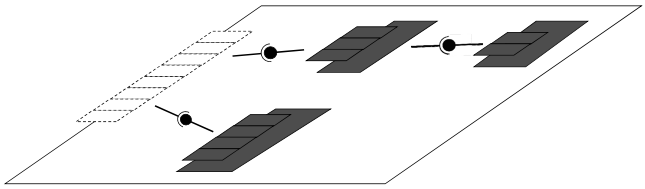

Figure 1. Exemplary flat component model.

The system (outer shape) is composed of three black box components held together by some special "glue code". The operations of the system have no functionality themselves (hence they are dashed) but present an aggregated interface to the collective functionality of the components.

More recent component models such as SOFA or Fractal [9] allow components to be nested in arbitrary ways across an unlimited number of levels. However, they still allow only the primitive components at the leaves of the resulting composition trees to contain rich functionality of their own. All the other components in the hierarchy, up to the system itself, are regarded as assemblies of lower-level components without any real functionality of their own. Their role is to combine lower level components together and present an aggregated interface to their functionality. These component approaches therefore retain the strict distinction between glue code and normal implementation code, with the latter still being used to implement the black box components at the leaves of composition trees, and glue code being used at the other levels to connect components together. We refer to such component models as **non-uniform, hierarchic component models**. This is illustrated in Figure 2.

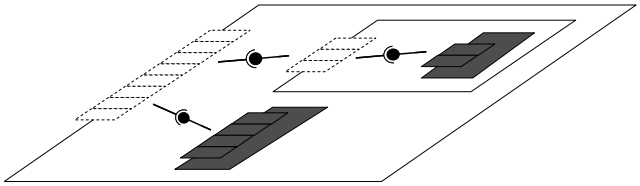

Figure 2. Non-uniform, but hierarchic component model.

The system is still composed of components, plugged together using special glue code, but these components may internally contain other components of their own. However, all the components except those at the leaves of the tree have "virtual" operations with no direct functionality of their own (and thus are still dashed in Figure 2).

Service technologies and standards [8] also generally treat primitive components (i.e. services) as black boxes that are implemented using mainstream programming technologies. For example, the web service standards only define the interface to services, and say nothing at all about their internal implementation. However, languages for composing services (such as BPEL) usually provide algorithm-definition features akin to those in programming languages and often allow the resulting functionality (or so called orchestrations) to be regarded as higher level services in their own right. General service models therefore allow services to be nested in arbitrary ways, and allow all services in a composition tree to have non-trivial functionality of their own, not just the primitive components at the leaves of the tree. We call such component models **uniform, hierarchic component models**. This is illustrated in Figure 3 where all components in the hierarchy, including the system itself, can have concrete methods with their own functionality.

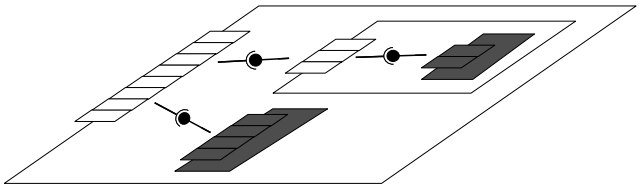

Figure 3. Uniform and hierarchic component model.

The differences between the various component models essentially revolve around the way they distribute functionality between components and allow them to be composed rather than on the organizational and procedural concepts used to develop systems. In other words, traditional component-technologies focus on the **architectural** aspects of software engineering rather than on the **process** of their development.

The only major process innovation offered by components is the notion of development by assembly [1] where systems are

developed by assembling existing parts rather than by traditional implementation techniques. However, this is still a far off vision. Although component discovery technologies have significantly improved in recent years [5] there are still significant barriers to software reuse such as the "not invented here syndrome" and licensing constraints. Thus, the effort / risks involved in finding and evaluating reusable components still usually outweighs the potential advantages [14].

3. MODEL-DRIVEN, COMPONENT-BASED DEVELOPMENT

As mentioned in section 1, several methods have attempted to merge the benefits of component-based development with model-driven development. Two of the earliest were the Catalysis [15] and UML components [16] methods which essentially defined approaches for describing systems and components using the UML. In effect, therefore, they combined a flat component model with the UML's concrete syntax. The most comprehensive merger of the two paradigms has been achieved by the KobrA method [11] which integrates the notion of uniform hierarchic component architectures with the three characteristic abstraction levels of model-driven development (i.e. Computation Independent, Platform Independent and Platform Specific [10]). Once again the UML was used as the concrete syntax for the CIM and PIM view of a composition hierarchy, but this is not important for the remainder of this paper.

As illustrated in Figure 4, in KobrA all architectural elements are modeled using the same set of views regardless of their size or location in the composition hierarchy. Figure 4 shows a system, S, composed of three components, A, B and C. A and B are direct subcomponents of S while C is a direct subcomponent of A. Each component (and the system, which is also regarded as a component), is represented by a cuboid surrounded by a collection of "views".

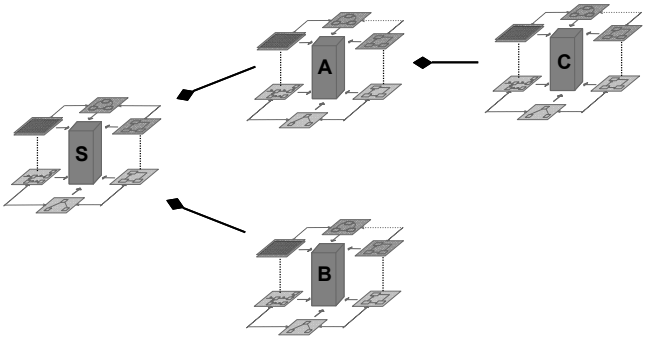

Figure 4. PIM Level, hierarchic component model.

Each component has a set of specification models and a set of realization models. The views around the top of each cuboid are the "specification" views that describe what services the component offers to its clients (i.e. other components that call its operations) and take the form of UML class diagrams, state diagrams and OCL pre- and post-conditions. The views around the bottom of each cuboid are the "realization" views that describe how the component realizes the specified functionality and what services it uses from other components in the system. They take the form of UML class, activity and collaboration diagrams. The key idea is that all components in the system, including the system itself, are treated uniformly (i.e. they are modeled in the same way) and they can all possess functionality of their own. In other words, all operations of S, A and B can contain rich algorithmic content. Amongst other component modeling approaches, KobrA has recently been applied to the Common Component Modeling Example (CoCoME) so that more details on its application are available [12].

4. I&I DEVELOPMENT

The top innovations in software engineering processes in recent years have mainly come from the agile development community in the form of lean, iterative and incremental development methods [17]. Iterative and incremental development also forms the backbone of more heavyweight methods such as the Unified Process [4] and is thus used in the majority of modern industrial software development projects. The reason is clear - by dividing a project into multiple mini projects, each adding a new increment of completed software to the code base over time, incremental development avoids the "big bang" integration and rigidity found in traditional waterfall processes. In agile projects, software applications can evolve gradually over time and all stakeholders can receive continual feedback on a project's status.

However, the enhanced leanness and flexibility of I&I development approaches comes at a price. Their emphasis on implementation and early delivery of working [17] code is in tension with the need to design a comprehensive, well-thought out, loosely-coupled architecture that takes all requirements into account. This results in software systems that in the worst case essentially have no architecture [6], or in the best case have a simplistic architecture in which software implementation elements are grouped into "layers" focused on technological aspects (e.g. GUI, business logic and persistence [4]). Genuine responsibility-oriented components of the kind making up component- and service-based systems are not recognized in mainstream I&I development approaches. Moreover, whenever an architecture eventually does emerge, it is often introduced post hoc in a series of expensive "refactoring" steps [13].

The problem with today's mainstream agile methods in this regard is that they are fundamentally function-oriented. They basically use functional criteria (based on the notion of use cases or user stories) to divide a software development project into multiple mini-projects that can be tackled incrementally. However, increments defined by functional slices through a system are usually orthogonal to the increments encapsulated by responsibility-oriented components[1] or technology-oriented layers. This phenomenon is illustrated schematically in Figure 5, which shows a system with a set of eight operations (small rectangles) invoked in various ways from three different use cases.

[1] By responsibility-oriented components we mean components that are focused on delivering a cohesive service or managing a specific sub-responsibility such as CRM, currency conversion, inventory etc.

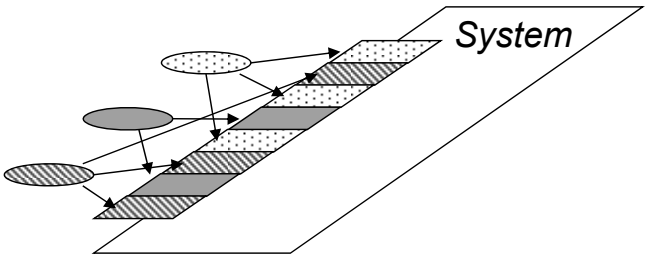

Figure 5. Use-case based system specification.

Figure 5 depicts the kind of system specification typically resulting from a model-driven, use-case-centric analysis of a system as e.g. proposed by the widespread RUP and Agile Modeling approaches [4]. Figure 6 shows how a layered implementation of such a system might evolve via the use-case-based increments typically applied in agile methods today.

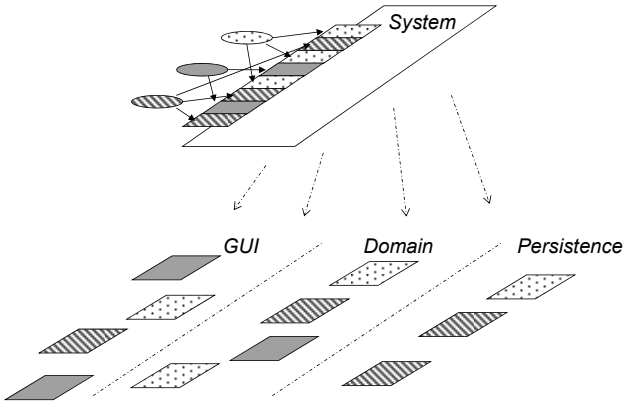

Figure 6. Implementation with layered architecture.

The elements in the lower level of Figure 6 are meant to represent software implementation units (e.g. classes) organized in a typical layering scheme (e.g. GUI layer, domain layer, persistence layer) [4]. For example, the first increment, driven by the striped use case, elaborates the striped units, the second increment, driven by the green solid use case, elaborates the solid units and so on. The problem is that the functionality encapsulated by use-case-driven increments is scattered among the units of the system in arbitrary ways (usually as objects) that do not match a component-oriented architecture. In other words, the currently most widely used approaches for systematic software development still lack support for a systematic definition of components.

5. COMPONENT-ORIENTED INCREMENTS

Although today's iterative methods almost always use functional criteria (e.g. use cases or user stories) to determine functionality increments to be implemented in individual cycles, this is not a fundamental requirement of incremental development. The defining characteristic of incremental development is the delivery of a functional and tested part of the system in each cycle. And for a method to be iterative all the main software engineering activities (analysis, design, implementation and validation) need to be performed in each cycle. How the increments and iterations are determined is basically immaterial. Therefore, provided a suitable hierarchical component model such as KobrA is available that supports platform independent descriptions of nested component architectures, it also becomes possible to define increments in a component-oriented way. While function-oriented increments tend to visit the classes making up the final software implementation tree in a depth-first way, component-oriented increments tend to visit them in a more breadth-first way.

This idea is illustrated schematically in Figure 7, which realizes the same system as Figure 6, but using a breadth-first implementation strategy with component-driven increments targeting a service-oriented implementation instead of the common depth-first, function-oriented approach. The same analysis models of the system appear on the left hand side of the upper level, but now these have been elaborated into a complete component-based system model in which the sub-components and sub-subcomponents of the system can be also modeled in a uniform and platform independent way (in the spirit of the OMG's Model-Driven Architecture [10]).

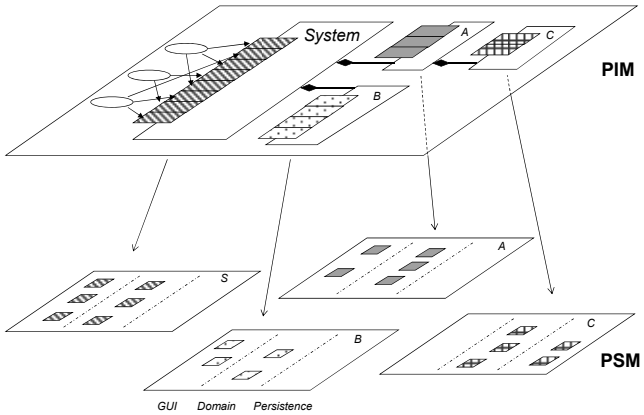

Figure 7. Elaboration of component-based increments.

Again the patterns show the increments of functionality that are implemented in each cycle. This time the first, striped increment is not driven by a use case but by the functionality implemented by the operations of the component providing the interface to the system. Once the requirements and realization of this "striped" functionality have been described at the platform independent (PIM) level (akin to analysis and design) the first development iteration is completed on the platform specific (PSM) level where it elaborates the striped implementation code. This might typically be based on some kind of service technology, internally organized using the common layered approach described before in which the functionality encapsulated by the system's operations is mapped to some kind of "orchestration" code. The next increment, the dotted increment, is then elaborated in exactly the same way, but this time driven by the functionality directly encapsulated by the operations of component B. Again, once the specification and realization models of the component (and its operations) have been completed, the corresponding implementation can be elaborated at the platform specific level. The solid and tiled increments are then elaborated in the same way.

Since all the key steps in software engineering are performed in each development cycle the approach is genuinely iterative. And since the delivered functionality is a complete, working piece of the final system the approach is genuinely incremental. The central difference to traditional iterative approaches is the way in

which the functionality addressed in increments is organized. Instead of driving the implementation based on functional considerations, the implementation is driven by the component-based architecture. Figure 8 shows the key artifacts involved in a typical increment when the approach is applied using the KobrA method.

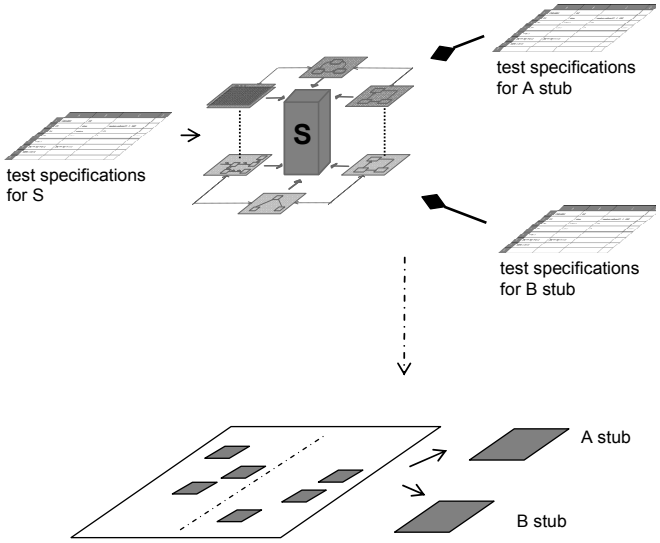

Figure 8. Artifacts (views) involved in a typical increment.

At the PIM level, the main artifacts generated in a component-oriented increment are the standard KobrA specification and realization views of the component under consideration (the so-called *subject*) as well as test specifications reflecting them. Component-based increments therefore also support the agile mantra of writing tests for an element before implementing it (i.e. test-first development [3]). The PIM level of a component increment also includes partial test specifications for the subcomponents used by the subject of the increment. These are not (yet) full tests for the components – they are specifications of the behaviour the subcomponents are expected to exhibit when invoked as part of the tests of the component.

The PSM level of a component increment contains implementations of –

1) the tests of the components, based on the test specifications
2) the component's functionality based on the realization models
3) stubs for each of the subcomponents based on their partial tests specifications.

Once all of these have been implemented, the functional code on this level (2) can be fully tested before being added to the delivered code base. Moreover, when the lower level subcomponents are implemented in subsequent increments, the component tests (1) can be reapplied with the real components rather than the stubs. The approach therefore also provides inherent support for integration testing.

6. DISCUSSION

For systems that lend themselves to a uniform, hierarchic component model, component-based incremental development appears to offer numerous benefits over standard feature-oriented agile development. It retains all the advantages of agile development (incremental evolution, iterations, test-driven development etc.) but addresses some of its core problems, namely –

First, it restores proper consideration of architecture into agile processes. Rather than being an afterthought that often emerges in a post hoc way, architecture plays a fundamental upfront role in the development process. Furthermore, (de-)composition decisions at a given level in the component hierarchy are made with a full knowledge of the requirements that the higher level components are required to satisfy.

Second, the improved, upfront consideration of architectural concerns is likely to significantly reduce the level of refactoring [13] that has to be performed in a development project. This is a major "Achilles heel" of traditional, function-oriented agile methods, and not only increases immediate development effort, it can have a residual impact on the quality of the final code, and thus ultimately on subsequent maintenance activities.

Third, the incremental development of component specifications and tests at the PIM level can significantly boost the chances of finding suitable pre-existing components before they are self-implemented, and thus could finally start to deliver on the promise of software reuse [1]. Since a full specification of a component is created before its design and implementation, this can be used as the basis for searching for existing components that already fulfill those requirements. Finding reusable implementation units at the PSM level is much more difficult because many more design decisions have been made and the room for flexibility is significantly reduced. In fact, even the test specifications developed to describe the required properties of subcomponent stubs are useful for attempting to find reusable components since they provide exactly the kind of input used by the recent generation of test-driven search engines such as Merobase [5] (i.e. they serve as representative descriptions for the expected behavior of the desired component [18]).

One apparent disadvantage of the component-based incremental development approach is the extra overhead involved in writing test specifications and stubs for subcomponents. It is certainly true that traditional agile methods do not involve such artifacts as they are not aware of components. However, they do require the development of other forms of stubs and test drivers to stand in for parts of the system that have not yet been built. Moreover, the effort involved in defining the subcomponent stubs has to be offset against the significant reductions in refactoring effort that can be expected from the requirements-aware architectural design just described.

Another disadvantage, inherent to all component- and service-oriented approaches, is the risk of duplicating functionality in various components. Suppose, for example that the green (A) and the blue (B) component in Figure 7 share some common functionality. Since A and B are supposed to be deployable independently of each other any common data structures or functionality need to be implemented in both. The natural way of dealing with this challenge in component-based development is to identify such elements and make them available as an additional

required component of A and B alike. However, to our knowledge, no component-based development method currently provides a systematic approach for performing this task. Only the techniques available in KobrA [11] for identifying the commonalities within software product lines seem to have potential for this purpose.

7. CONCLUSION

In this article we have proposed a way of reconciling component-driven development approaches with modern, I&I development methods that creates a powerful synergy between them. Components primarily focus on architecture, and have traditionally been neglected in mainstream development approaches, especially in agile methods. I&I methods, on the other hand, primarily focus on implementing function-oriented (i.e. use case or feature-oriented) slices of functionality and have so far generally neglected responsibility-oriented components of the kind used in component- and service-oriented architectures. In other words, the architectural "Achilles heel" of agile methods is the main strength of component technologies, while the process "Achilles heel" of component technologies is the main strength of agile methods.

The key to the synergy discussed in this article is to introduce a platform-independent model of the architecture based on a uniform, hierarchic component model that allows the functionality in the system to be distributed among all the components in a system, not just those at the leaves of the component hierarchy. This in turn allows increments of functionality to be elaborated in a component-oriented rather than a function-oriented way. Although this radically departs from accepted ways of organizing agile development projects it remains faithful to the key tenets of incremental and iterative development. Furthermore, the approach is able to retain all other key aspects of agile development such as the up-front development of tests before implementation.

The problem of finding the optimal component-based architecture for a system (i.e. of identifying the right components and component configurations) is a challenging one. We make no claim that the approach described in this position paper helps in this regard, except that it allows decisions about subcomponents to be made in full knowledge of higher level requirements. In domains where traditional agile methods are currently used, it is difficult to say whether the domain-oriented, hierarchical component architectures advocated in our approach are better than the layered architecture and "no architecture" approaches usually associated with agile methods today. But in domains where responsibility-driven components are a natural way of (de)composing systems, organizing the development process in terms of component-oriented increments appears to offer many advantages. And in domains where the use of components and/or services is well established, the ability to leverage agile principles in their development is likely to significantly enhance the way such systems are engineered and evolved.

8. REFERENCES

[1] McIlroy, D.: Mass-Produced Software Components, Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 1968.

[2] Szyperski, C. Component Software: Beyond Object-Oriented Programming (2nd ed.), Addison-Wesley, 2002.

[3] Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999.

[4] Larman, C. Applying UML and Patterns (3rd ed.). Prentice Hall, 2004.

[5] Hummel, O., Janjic, W., Atkinson, C. Code Conjurer: Pulling Reusable Software out of Thin Air, IEEE Software (Vol. 25, Iss. 5), 2008

[6] Booch, G. The Accidental Architecture. IEEE Software, Volume 23, Issue 3, 2006.

[7] Crnkovic, I., Chaudron, M., Larsson, S. Component-Based Development Process and Component Lifecycle. Proceedings of the International Conference on Software Engineering Advances, 2006.

[8] Erl, T. Services-Oriented Architectures, Prentice Hall, 2005.

[9] Lau, K.K., Wang, Z. Software Component Models. IEEE Transactions on Software Engineering, Volume 33, Issue 10, 2007.

[10] Kleppe, A., Bast, W., Warmer, J. MDA Explained: The Model-Driven Architecture. Addison-Wesley, 2003.

[11] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-based Product Line Engineering with UML, Addison-Wesley, 2001.

[12] Atkinson, C., Bostan, P., Brenner, D., Falcone, G., Gutheil, M., Hummel, O., Juhasz, M., Stoll, D.: Modeling Components and Component-Based Systems in KobrA. In Rausch, Reussner, Mirandola, Plasil (editors): The Common Component Modelling Example, Springer, 2008.

[13] Fowler, M. Refactoring, Addison-Wesley, 1999.

[14] Sherif, K., Vinze, A. Barriers to adoption of software reuse: A qualitative study. Journal of Information Management, Vol. 41, No. 2, 2003.

[15] D'Souza, D.,Wills, A. Objects, Components, and Frameworks with UML: The Catalysis Approach: The Catalysis Approach, Addison-Wesley, 1998.

[16] Cheesman, J., Daniels, J. UML Components: A Simple Process for Specifying Component-Based Software, Addison-Wesley, 2000.

[17] Schwaber, K. Agile Project Management with Scrum, Microsoft Press, 2004.

[18] Hummel, O. Semantic Component Retrieval in Software Retrieval, PhD Dissertation, University of Mannheim, 2008.