

Model Driven Development of Component Centric Applications

Andreas Heberle (entory AG), Rainer Neumann (PTV AG)

Abstract. The development of applications has to be as efficient as possible. The Model Driven Architecture (MDA) is one way to achieve efficiency but it requires a complex and expensive tool set. In this paper, we present an approach that combines the benefits of the model driven architecture with component centric development. The approach is lean, cheap, extendable, and strictly component-centric. It comes along with a sound component model, a model for the composition of components and the definition of business processes, a process model, and a small generator and design tool set. So the approach can be applied in every days project business. Because the approach is platform independent it can be used in the J2EE as well as in the .Net or any other technological area.

Introduction

Despite component orientation development of complex software systems lacks still of efficiency and has to deal with quality problems.

In our opinion, component-centric development of software should be as lean and flexible as possible. It should be supported by a tools set that allows for choice of different architecture styles and usage of specific design and implementation patterns. Platform independence is also an important point. In this paper we present the approach developed in a research project called CompoBench which aims to develop tools for component oriented software. The presented work focuses on business applications. Basis of the approach is a sound component model and a model for the composition of components. Within the latter business processes are specified composing activities. The notation for models is the Unified Modeling Language (UML) [3]. The approach is supported by a generator that takes the XML Metadata Interchange (XMI) [5] representation of a UML model and generates platform specific code. The generator uses the component model to produce the component skeletons, the business process model is used to generate workflow and composition code. The generation process uses implementation templates and is highly configurable with respect to design and deployment aspects. The generator can be easily extended or modified. Within CompoBench the approach is embedded in a iterative software process model and is only one tool in a bunch of tools.

In this paper we first give a rough overview to the development process. We introduce the composition model and then describe the component model. After that

we present the generator and talk about our experiences in projects the generator is already in use.

Related Work

There exist several tools and approaches for model driven development of software. The most popular approach is the model driven architecture (MDA) [1]. However, this approach has several drawbacks. Using components as black boxes is not really supported since an accurate component model is missing. A sound composition model which is the pre-requisite for composing applications from components does not exist either. The MDA is quite complex, existing tools are expensive and their usage requires an intensive training.

Another group of tools like UML2EJB [6] and its successor AndroMDA [7] are open source. They follow a similar implementation approach based on UML as the modeling language, templates and Xdoclet [8] than we do. But, like the MDA, they also suffer from the absence of component and composition models. All of these tools generate only component skeletons, no workflow code.

Other tools like e.g. Aris [9] support business process management. From the business process specifications code can be generated also. The tools are process-centric and not component-centric.

In the context of web services new languages for the specification of business process behaviour and the interaction of web services were introduced. The Business Process Execution Language for Web Services (BPEL4WS) [10] and the Web Service Choreography Interface (WSCI) [11] might become a standard and thus might influence the definition of our composition model.

Component Centric Development

The presented approach is influenced by three different requirements that are in our opinion essential to an efficient and systematic development process:

1. Adequate Specifications – Business analysis and design should be done by people knowing the business, and not by technicians, guessing what the system may be. This means that the specification of a system must be done on an adequate abstraction layer.
2. Adaptable Code Generation – There should be a mostly automated way to derive implementation code from business code wherever feasible. This does not mean that each sorting or file reading function needs to be specified on a high abstraction layer but that high level specifications must be mixed with lower level code.
3. Modularity – While the first two requirements are part of the MDA idea, that approach does not care about modularity, which is the core concept for component oriented software. To overcome this gap, we propose a high level component model in the following sections. But before describing that model, we first introduce the development process, how we see it.

Component centric development consists of the following development steps:

1. Modeling of business cases, business processes and business modules
2. Design, identification, and adaptation of components
3. Implementation of activities and composition of components to implement business processes
4. Parameterization of generation
5. Code generation

In this approach, the first step deals with business entities only, while the second step bridges the gap from business modeling to implementation. In our opinion, this second step can be done systematically having an adequate component model and an according composition model. The definition of components and applications is business driven. This means in the beginning, business objects, e.g. account or share in the banking area, and the business use cases and processes like opening an account or trading a share, have to be defined by the business analysts.

To our experience, the essential part in the development process is to find activities and to model processes – the latter is quite simple having found the activities. To overcome this, we propose the following design approach:

1. Define the high level business processes. There are usually several business roles (actors) involved in process execution (such as customers, employees, or system processes)
2. Each actor taking part in a business process performs several activities. We now map these high level activities to modules of the system (we found swimming lanes pretty useful on that level of abstraction). Now each of the activities defines a business use case that consists of sequences of system interaction and/or automated process parts.
3. Finally, the business cases (the high level activities) are split into parts that are triggered by the according actor (this is usually the synchronous part of the control flow). These new fine grain entities are the activities that we can use to refine the process model. These fine grain activities are part of the components interfaces. Hence, mapping a new process to an existing component infrastructure takes place on that level of abstraction.

Having described the overall design process, we now illustrate the composition and the component model that support the process in the following two sections.

The Composition Model

The originator for the composition model is the business process. A business process consists of several steps so called activities. The process model is hierarchical. An activity can either be atomic or it can itself be a process that consists of activities. An activity has a specific beginning and a specific termination point. During processing an activity data objects will be created or existing data objects will be modified. Furthermore events may be raised that are signals other activities can react on. The static aspects of activities are described in more detail in the next section.

The composition model defines the dynamic aspects of activities. It describes control and data flow over the activities of a process.

Control Flow

Activities can be processed sequentially or in parallel, they can be activated conditionally. Furthermore activities have to be synchronized and react on events created by other activities. The composition model provides appropriate control flow elements. Modeling control flow means connecting activities by these control flow elements.

Data Flow

An Activity creates new data objects and modifies or deletes existing data objects. The composition model provides the appropriate elements to model data flow. Data flow between activities is specified connecting input and output objects of activities.

The following figure shows an example for the composition of activities that uses the elements of the composition model.

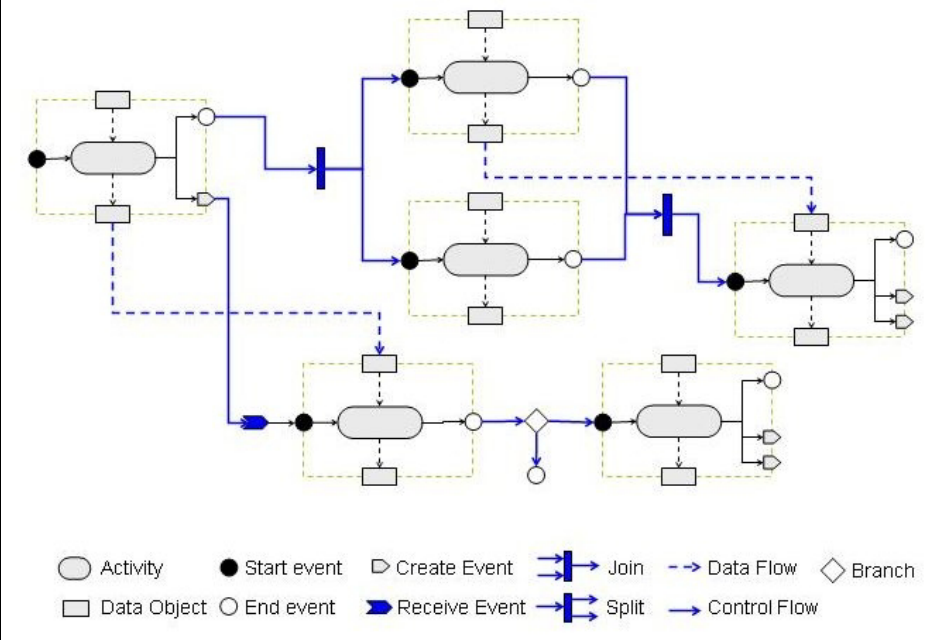


Fig. 1. Process centric composition

Though in the figure above we used a special graphical notation, business processes can be specified by UML activity diagrams. A special graphical notation for processes, data and control flow might be desirable but is not necessary. There exist similar notations, e.g. the one defined by Eriksson and Penker [2].

Having defined the business model, the activities have to be mapped to components. For this binding the static aspects of a component are important: interfaces, methods, data objects. Each business activity has to be mapped to a component activity, the data objects in the composition model have to be mapped to the component model's data. A component may implement several activities, a process may be implemented by several components.

Components can be composed horizontally and vertically. Horizontal composition describes the composition of components of the same architectural layer. Here, composition is implemented by means of the implementation model, e.g. in J2EE by using sequential component calls, JMS and message driven beans. In order to bridge differences in interfaces or to implement loose coupling adapters and mediators are used. Vertical composition means the composition of components between different architectural layers, e.g. between the application layer and the presentation layer. With this kind of composition one often has to bridge technical gaps, e.g. the integration of J2EE and .Net components. Adapters or web services are examples for the implementation of vertical composition.

The Component Model

On the one hand the component model is influenced by the requirements of the composition model. On the other hand reuse and business requirements have implication to this model.

Having described the dynamic parts in the composition model we now come to the static parts that make up component specifications. We here focus on the interface of a component - the implementation model is described in the following section.

The interface of a component consists of the following elements:

Data Model

This data model specifies objects and relations accessible via component functionality. A relation between two objects means that there is a possibility of traversing the relation from one object to another using component functionality. At this point it is irrelevant whether the relation is implemented by a database relation or computed ad-hoc by a function. However, at the interface level we have to distinguish between several kinds of data objects, mainly persistent and transient objects.

Usually, relations between persistent data objects will be implemented by database relations, but they can also be derived relations that will be computed from others.

Business Functions

These functions are defined by specifying one or more interfaces which contain methods that represent the business functions. The business interfaces can be compared with service descriptions, such as web services or interfaces of session beans in a J2EE context.

Activities

The third and last part of the component interface are the activities. These are special implementation parts used for process modeling as described in the previous section.

Activities are seen as functional nodes with the following properties:

- **Activation**

An activity is started by a control flow event. Connecting the activity to an event trigger is part of the process model. In this case an activity is passive until it is started / activated.

- **Deactivation**
An activity may run forever. However, most activities will finish after a certain amount of time. This is called the deactivation or finish of the activity.
- **Data Flow**
An activity reads data and writes data. Data can be read en-bloc at activation time (written at deactivation time respectively), or it can be read from a data stream (written to a stream respectively). The data flow kind is part of the activity specification.
- **Events**
During its life-time, an activity can raise several events, e.g. if new data is created or deleted. Events are asynchronously signaled, while control flow remains within the activity.
- **Hooks**
Hooks (a.k.a. callbacks) are similar to events but follow a synchronous activation model. This means, the activity is passivated until the hook has been handled. This is required to implement add-ins for generic processes and control flow frameworks, where additions can be plugged in easily.
- **Protocols**
Activation, deactivation, events and hooks follow sequences which can be described best using state charts. These protocols make up an integral part of the activity specification.

The figure 2 shows the principle layout of an activity specification.

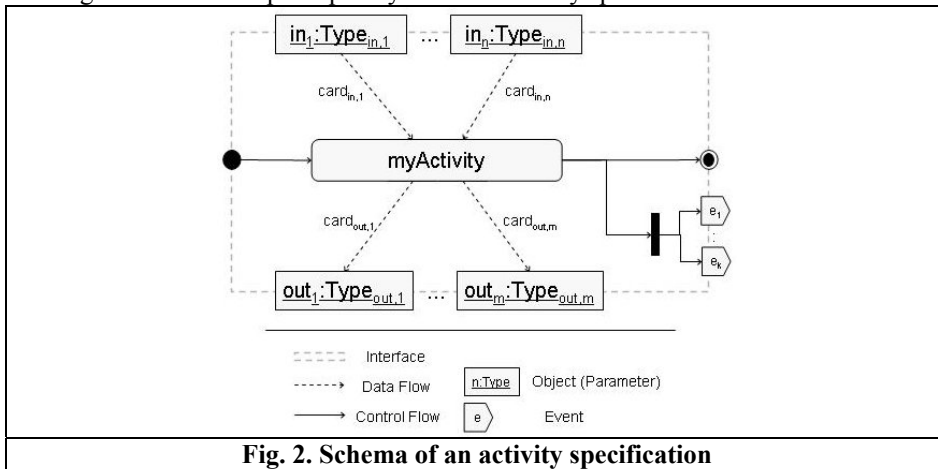


Figure 3 shows an example of such a course grain component consisting of a data model, business interfaces and activities. This model extends the traditional component model, e.g. as known from Enterprise Java Beans or CORBA components: Having a notion of a complex data model and activities allows for encapsulation and reuse on a much higher level of abstraction. However, reusing such a rich component may require a high degree of configuration and adaption, which has partially been explained in the previous section. But many of the reuse, adaptation, configuration and composition issues still have to be proven practicable in the near future.

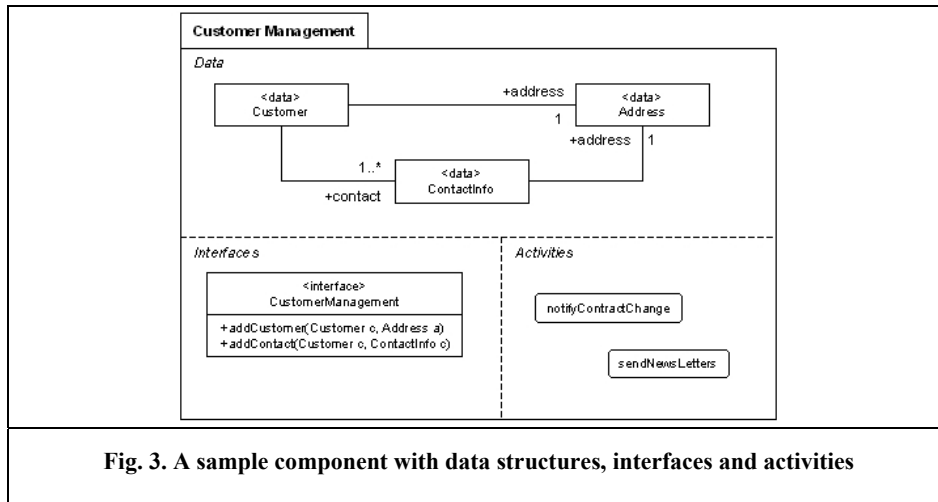


Fig. 3. A sample component with data structures, interfaces and activities

A Tool Set for Model Driven Development

Now that we roughly sketched the specification of components and their composition, we briefly describe the generation model and the tools that are used to get the whole thing running.

The main idea is to have a layered generation model to allow for specification on adequate abstraction levels. For instance, the body of a function is usually best expressed in terms of a programming language, whereas the data model of a component can easily be described in a UML model.

Hence, the work presented here uses a multi-layer approach for the generation, where a more abstract layer is transformed into a lower layer and then combined with specifications on that lower level. The figure 4 shows that layered generation model.

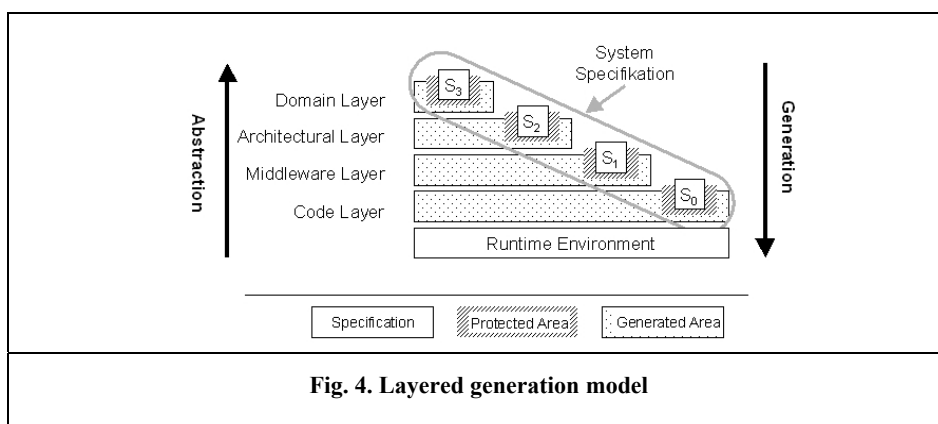


Fig. 4. Layered generation model

Since the explanation of the whole layer model would take too long, we here focus on the topmost layers, which are the business domain layer, the component design layer and the component implementation layer.

The top layer - the business analysis layer (not shown in the figure)- is used for domain analysis purposes. The terms used on that layer are mainly *business data*, *business process* and *business component*.

Transforming the business layer into the underlying domain (design) layer is a creative manual step. One has e.g. to decide whether business data is represented by persistent or transient data. So this step will be left out for now.

The domain design layer is where the tools come in. The layer is specified using UML diagrams, where the terms component, activity, interface, persistent data, transient data and relations are specified. Therefore we use stereotypes. The specification on the component design layer is used as the input for the generation tools, which are described shortly in the following.

The toolset we use for generation is a collection of small tools plugged together in an ant-based build process. The first part in the chain are the first level generator tools, which produce source code from a given UML model. These tools are followed by pretty printers or other second-level-tools, such as xdoclet that take the technology specific generation.

The first level tools are template based generators using a template engine, a UML modell access service and a set of templates. This approach is similar to the *uml2ejb* approach or its successor project *androMDA*. In fact we are discussing the move to the latter framework since the main focus of our approach is the design of the templates that implement the design.

The templates can be seen as patterns that control code generation for specific technologies. Hence, by exchanging or modifying the templates, we can easily generate code for new technologies or enhanced base frameworks.

Experiences and Evaluations

Until now we only use a generator for the static aspects of the component model. The component skeletons and the data model are generated from a UML specification. The generator is used in different projects. There components are generated for different architectures using different implementation patterns.

The first generation of the tools have already been used in mid-sized industry projects with great success. They showed, how the amount of code to be written can be reduced to a real small amount, when using complex base technologies such as J2EE or WebServices.

In fact, we got one mid-sized project out and running, where the static UML model has been used to generate everything starting from the server-side object model, the database mapping, the structure maintenance services, web-services for accessing the object structure, .NET clients accessing these web-services, client side caches for reduced network traffic, up to the .NET data browsing application.

From our current experience we can say that:

- Developers liked to use the generator since it simplifies the development and saves a lot of tedious work. However, at first they were - and hence we guess others will be - pretty sceptical at the beginning, but once they learned how to use the generators and especially how to adapt and extend the templates, the scepticism was gone and they started to be very efficient.
- Architects liked the generator since it improved the development speed and standardized the code. Knowing the standards makes it easier to understand the code implemented by other people and so allows for code share and helps developers being new in a project. The generated code heavily depends on the used code generation templates. Since these templates are easy to understand and modify, changes and improvements can be performed easily. Last but not least, business code could be clearly separated from the environmental code, which gave a pretty good feeling for code sizes and project complexity.
- Project managers liked the tool since it was for free and did not require lot of training.

Focusing on the model helped a lot in communication within the teams - especially between developers, architects and managers, which usually speak different languages.

Having integrated the tools to an ant based build environment makes it very easy to control build process from a standard development environment like eclipse.

Since we rely on the UML standard the specification of components and in future of processes and workflows can be defined easily by people being used to the UML notation. Furthermore, with the new extension of UML 2.0 [3], particularly for components, some of the extensions we made in our UML profiles become superfluous. So we will get closer to the standard notation.

Summary

The concept of component oriented software development already exists for several years. There exist component systems like CORBA, .Net and EJB together with runtime environments like application servers. Nevertheless, component oriented software development is not yet a success story. Complex .Net and EJB projects suffer from development efficiency and quality. One reason is the lack of smart, efficient and tailorable tools. One goal of the CompoBench project is to develop a framework and a set of tools for component oriented software development. The presented approach comprises some of these tools: the model for the specification of components and the composition of components and a generator that allows for automatic transformation of the models to code.

Our approach has several advantages:

- The composition model is based on specifications of the real business processes. So we bridge the gap between business and technical teams.
- There is a sound and exact model for the specification of components and the composition of applications.
- With UML the specifications are conform to a well established standard.

- There is a lean generator tool that is able to produce code for different platforms. The process can be easily adapted and extended by simply writing new implementation templates.
- Existing tools for modeling can be used since the generation works on the XML representation of UML models.

We presented in this paper ongoing work. Not all parts of the models are already implemented. The generation of composition code is still missing and has to be done by hand. In the near future we will close this gap - first prototypes are already being developed and evaluated. Nevertheless, we already used and are continuing to use the tools successfully in projects. We will also improve the models incorporating the experiences made in several case studies being part of the research project scope [4].

Bibliography

- [1] Object Management Group (OMG) 2002. *Model Driven Architecture*. <http://www.omg.org/mda/>.
- [2] Hans-Erik Eriksson and Magnus Penker. 2000. OMG Press. *Business Modeling With UML*. Business Patterns at Work.
- [3] Object Management Group. 2002. *OMG UML Home Page*. <http://www.uml.org>.
- [4] CompoBench Consortium. 2002. *CompoBench Project Homepage*. <http://compobench.fzi.de>.
- [5] Object Management Group (OMG). 2002. *XML Metadata Interchange (XMI), version 1.2*. <http://www.omg.org/technology/documents/formal/xmi.htm>.
- [6] 2003. *UML2EJB from UML to Enterprise JavaBeans*. <http://uml2ejb.sourceforge.net>.
- [7] 2003. *AndroMDA from UML to deployable components*. <http://www.andromda.org/>.
- [8] 2002. *Xdoclet - Attribute-Oriented Programming*. <http://xdoclet.sourceforge.net/>.
- [9] IDS Scheer AG2003. *Aris 6 - Collaborative Suite, White Paper*. <http://xdoclet.sourceforge.net/>.
- [10] 2002. *Business Process Execution Language for Web Services Version 1.1*. <ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf>.
- [11] 2002. *Web Service Choreography Interface (WSCI) 1.0*. <http://www.w3.org/TR/wsci/>.

We thank our colleagues in CompoBench for the fruitful discussions and the cooperative work.