IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com
ITP4412

Specification of Components Based on the WebComposition Component Model

Martin Gaedke[a] and Klaus Turowski[b]
a.Telecooperation Office (TecO), University of Karlsruhe, Vincenz-Priessnitz Str. 1, D-76131 Karlsruhe, Germany, E-mail: gaedke@webengineering.org
b.Business Information Systems, University of the Federal Armed Forces Munich, Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany, E-mail: turowski@informatik.unibw-muenchen.de

ABSTRACT

Developing application systems that use the World Wide Web (WWW, Web) as an application platform suffers from the absence of disciplined approaches to develop such Web-applications. Besides, the Web's implementation model makes it difficult to apply well-known process models to the development and evolution of Web-applications. On the other hand, component-based software development appears as a promising approach that meets essential requirements of developing and evolving highly dynamic Web-applications. With respect to Web-applications, its main objective is to build Web-applications from (standardized) components. Founded on these insights and based on a dedicated component model, we propose an approach to a disciplined specification of components.

1 WEBCOMPOSITION COMPONENT MODEL

The *WebComposition* component model (Gellersen & Gaedke, 1999) describes the way of composing Web-applications from components. It bridges the gap between design and implementation by capturing whole design artifacts in components of arbitrary granularity. The resolution of a component is not preset but can vary depending on the level of detail required by the design concept in question. A component may represent, e.g., an atomic feature such as the font size attribute, a complex navigation structure, implementations of hypermedia design-patterns, or simply compositions of other components. In this way, WebComposition supports the bridging of the gap between the design and the implementation model by offering a high-resolution implementation model relying on code-abstractions. We construct complete target language resources by compiling compositions of these components. In the following sub-sections we describe the WebComposition approach, which is based on the WebComposition component model. The complete WebComposition approach defines a disciplined procedure of composing Web-applications with components (Gaedke, 2000). It is a synthesis of a component-oriented process model with a dedicated Web-application framework, reuse management, and a dedicated component-technology.

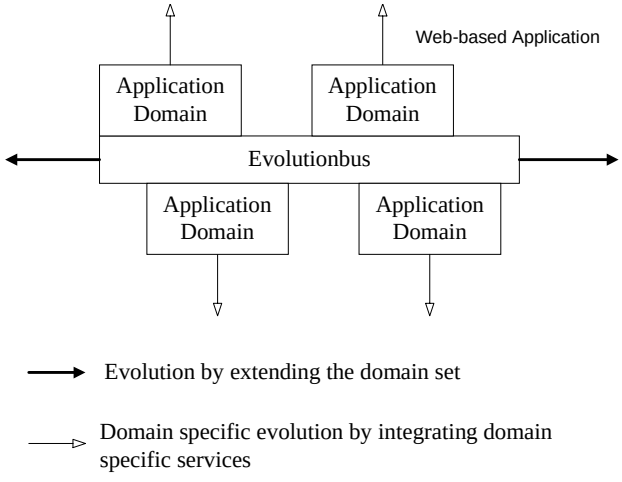

Figure 1: Dimensions of a Web-application's evolution space

1.1 WebComposition Process Model

The requirements for a software system change as time goes by. It is obvious that many kinds of influences are responsible for this, e.g. new regulations, changes in corporate identity or an extension of functionality. Such maintenance tasks are difficult to handle if we did not design the application with the possibility of later changes and extensions in mind.

The *WebComposition Process Model* focuses on the evolution of Web-applications by reusing components. It consists of three main-phases. The phases are derived from the common phases of (object-oriented) process models as well as solutions addressing the need of software reuse, and taking the principles of the Web into account. The process model follows a spiral consisting of evolution analysis and planning, evolution design and the execution of evolution. The first phase deals with common problems in strategic planning of the applications' functionality respectively with Domain Engineering. *Domain Engineering* has been described as a process for creating a competence in application engineering for a family of similar systems. The last two phases reflect the two different views towards reuse: consumer view (*development with reuse*) and producer view (*development for reuse*). We facilitate the evolution of an application following these phases by a framework, which maps the results of each phase directly to components. We explain the process in the following.

To allow for a disciplined and manageable evolution of a Web-application in the future it makes sense not to design the initial application on the basis of the concrete requirements identified at the start of the project. Instead the initial application should be regarded as an empty application that is suitable for accommodating functionality within a clearly defined evolution space.

This approach is based on domain engineering. During the analysis phase we determine the properties of an application domain. During the design phase we transform this information into a model for the application domain. From this, we can determine the required evolution space and, during the implementation phase of the domain engineering process, we can construct the initial application as a framework ready to accommodate any kind of functionality that lies within the evolution space of the domain.

This paper appears in *Managing Information Technology in a Global Economy*, the proceedings of the Information Resources Management Association International Conference.

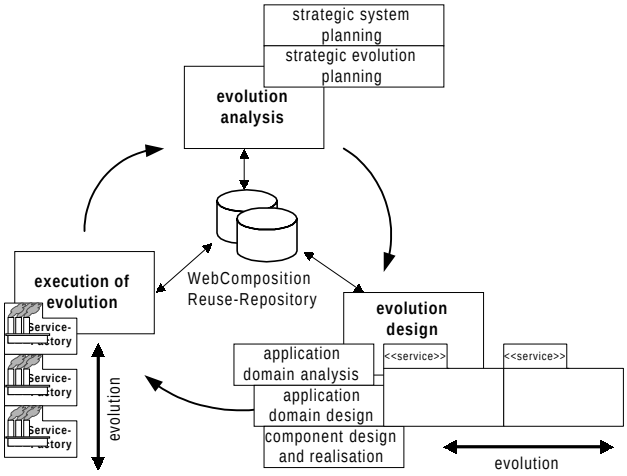

Figure 2: Complete evolution process

We can extend this view to several application domains. Therefore, we use the term *evolutionbus* for the basic architecture of a Web-application. The evolutionbus is the initial application for all abstract application domains of a Web-application, cf. Figure 1.

The evolutionbus enables the management and collaboration of domain-components, i.e. components that implement specific application domains such as Web-based procurement, reporting, or user driven data exchange. These domain-components (also called *Services* within the WebComposition approach) also represent prototypes for future services of the same application domain. The evolution can take place in two clearly defined ways:

- *Domain specific evolution (evolution design)* – The extension of a domain through new services, e.g. by prototyping an existing service of a domain. Another possibility is that the domain itself changes or that it receives more functionality, which requires the modification of the domain's initial service that serves as a prototype for other services.
- *Evolution of the domain set (evolution execution)* – The evolution of an application is also possible through the modification of the domain set. The extension of an application's functionality by adding a new application domain takes place, e.g. when a shopping basket and corresponding functionality is added to a Web-based product catalog. The integration of a new domain is realized by connecting a new initial service to the evolution bus (this mechanism can be facilitated using dedicated editors or automated by factories, cf. Factory design-pattern).

Figure 2 gives a detailed overview of the complete process.

1.2 Reuse Management

It is hoped that growing numbers of components increase the probability that a component fitting a certain purpose exists. On the other hand, the difficulty associated with finding such a component also increases with larger numbers of components. As soon as a lot of components are available finding appropriate components becomes one of the main problems of code reuse and of the CBSE (c*omponent-based software engineering*) approach especially. In short, this so-called *component dilemma* states that the probability to own a component that can be used to solve a specific problem increases with the number of available components while at the same time the effort needed to locate such a component within the set of available components increases as well. The retrieval of components in libraries is therefore a widely discussed problem. Component repositories can be an answer to problems posed by a situation in which a human developer cannot be acquainted with all of those components (let alone know all the details about them).

Repositories intended for reuse can employ different methods for the classification and representation of components to improve the chance of finding a component matching a given development problem and to present an augmented perspective of the stored components. The commonly used representation methods usually belong to (at least) one of the following categories: *controlled and uncontrolled indexing* or *methods that contain semantic information*. Also *hypertext-based systems* are mentioned sometimes.

In the WebComposition Process Model the *WebComposition Repository* is the responsible tool for the administration of reusable components (cf. Figure 2). There is no single program, which constitutes the repository. Instead, the basic mode of operation is the cooperation of at least three system entities: a component store, at least one *Metadata Store*, and a search or browsing tool *(Repository Tool)*. The tool can utilize the information stored in the Metadata Stores to provide advanced retrieval abilities or it can display information from the Component Store augmented with additional information provided by the Metadata Stores. We shape tools to work with the information of certain sets of Metadata Stores. Furthermore, we propose a disciplined approach to specify components in a consistent and reuse friendly way (see section 4).

1.3 Dedicated Component Technology for the Web

The *Web Composition Markup Language* (WCML) was introduced in (Gaedke, Schempf, & Gellersen, 1999) to offer a convenient way to define and represent components. WCML is an application of the *eXtensible Markup Language* (XML) and allows a (tag-based) definition of components, properties, and relationships between components on top of WebComposition's object-oriented prototype-instance-model (Gellersen & Gaedke, 1999). As an application of XML the WCML is platform independent, easy to parse, and it is rigorous in terms of well-formed or valid documents.

Within the WCML model we describe a Web-application as a composition of components. From the perspective of the progress of different processes, Web-applications can consist of a hierarchy of components that each correspond to whole parts of Web-applications with resources or fragments of resources. On the other hand, for certain parts of a Web-application only information from analysis or design may exist. Components within the WCML-model are identifiable through a *Universally Unique Id* (UUID). They contain a state in the form of typed attributes, called properties, which resemble simple name-value-pairs. Further, the value of a property can be defined through static text or WCML language constructs and must correspond to the data-type of the property.

The before mentioned concept of developing component software by composition is realized with WCML language constructs. Each component can be based on any number of other components and use their behavior by referencing them or by using them as prototypes. A modification of a component can therefore consistently change all components that use it.

2 SPECIFYING WEBCOMPOSITION COMPONENTS

To store components in a repository and to further retrieve and reuse them, we have to describe their interface and behavior in a consistent and unequivocal way. In short, we have to *specify* them. *Software contracts* offer a good solution to meet the special requirements of specifying components. Software contracts go

back to MEYER, who introduced contracts as a concept in the *Eiffel* programming language. He called it *programming by contract* (Meyer, 1988).

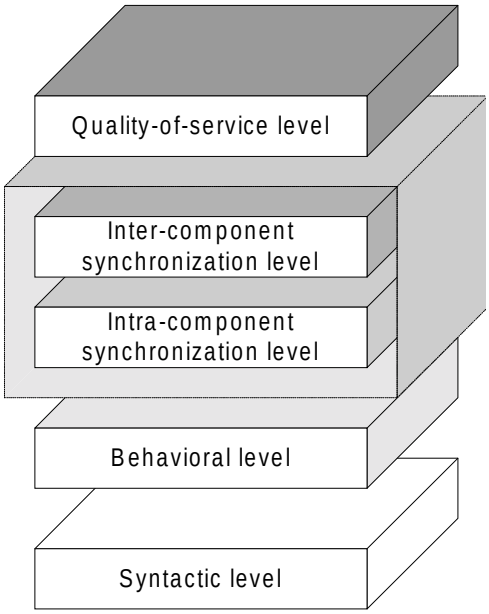

Figure 3: Software contract levels

Software contracts are obligations to which a service donator (a component) and a service client agree. There, the service donator guarantees that

- a service it offers, e.g. calculate balance or determine demand,
- under certain conditions, which have to be met by the service client, e.g. the provision of data necessary to process the service,
- is performed in a guaranteed quality, e.g. with a predetermined storage demand or with an agreed response time, and
- that the service has certain external characteristics, e.g. the specified interface.
- Figure 3 shows contract levels according to (Turowski, 1999). At syntactic level, we conclude basic agreements. Typical parts of these agreements concern names of services (offered by a component), names of public accessible attributes, variables, or constant values, specialized data types (in common based upon standardized data types), signatures of services, as well as the declaration of error messages or exception signals. To do so, we use e.g. programming languages or ***Interface Definition Languages*** (IDL) like the IDL that was proposed by the Object Management Group (OMG). The resulting agreement guarantees that service client and service donator can communicate with each other. With this, we put the emphasis on enabling communication technically. Semantic aspects remain unconsidered.

Agreements at behavioral level serve as a closer description of a component's behavior. They enhance the basic agreements of the syntactic level, which mainly describe the syntax of an interface. Agreements at syntactic level do not describe how a given component acts in general or in borderline cases.

As an example, we could define an invariant condition for a component *stock keeping* at behavioral level, which says that the reordering quantity for each (stock) account has to be higher than the minimum inventory level. Known approaches to specify behavior are based on approaches to *algebraic specification* of abstract data types, cf. e.g (Ehrig & Mahr, 1985). To describe behavior, we extend the specification of an abstract data type by conditions. These conditions describe the abstract data type's behavior in general (as *invariant conditions*) or at specific times (*pre conditions* or *post conditions*). In general, conditions are formulated as equations, and as axioms they become part of the specification of an abstract data type. The *Object Constraint Language* (OCL) (Rational Software et al., 1997) is an example for a widespread notation to specify facts at the behavioral level. It complements the *Unified Modeling Language* (UML).

Agreements at intra-component synchronization level regulate the sequence in which services of a specific component may be invoked or navigated to, and synchronization demand between its services. Here, e.g., we may lay down that a minimum inventory level has to be set before it is allowed to book on a (stock) account for the first time, or that it is not allowed to carry through more than one bookkeeping entry at the same time for the same account.

At inter-component synchronization level, we come to agreements that regulate the sequence in which services of *different* components may be invoked. Here, e.g., we may define that a certain service, which belongs to a component *shipping*, and which refers to a certain order, may only be processed after a service, which belongs to a component *sales*, and which refers to the same order, has been processed at any time before.

There exist various approaches to specify components at the synchronization levels. These approaches base, e.g., on using *process algebras*, *process calculi*, or on using *temporal logics*. In addition, (semi formal) graphical notations are in use, e.g. Petri net-based notations.

As an extension to functional characteristics, we have to describe *non-functional* characteristics of components. Non-functional characteristics are specified at the quality-of-service level. Examples for these characteristics are the distribution of the response time of a service or its availability.

We propose to use the OMG IDL or WCML-prototypes (syntactic level), the UML OCL (behavioral level), and the UML OCL with temporal extension (inter- and intra-component synchronization level) to specify WCML components. At the quality-of-service level we so far use natural language. These specifications are encapsulated in standardized XML markup to ease its use in the Web environment.

Figure 4 shows an example of how we specify the process component *PrintInvoice* as part of a WCML service *OrderProcessing* at behavioral level. There, we use a pre condition for the component *PrintInvoice*. It ensures that printing an invoice is allowed, if and only if the corresponding order was delivered before. Furthermore, there is a post condition that explains in detail, how the invoice amount was calculated.

```
<Service>
  <UUID>OrderProcessing</UUID>
  <ProcessComponent>
    <UUID>OrderProcessing</UUID>
   <BehavioralLevel>
    <Method>
     <Name>PrintInvoice</Name>
      <Signature><Name>at</Name><Type>Order</Type></
Signature>
    </Method>
    <OCL>
      <PRE>
```

```
        self.Order->exists(a:Order | a = at and at.Delivered =
True)
      </PRE>
      <POST>
        at.InvoiceAmount =
          at.OrderPositions->iterate(p:Position; b:Amount = 0 |
          b + p.Quantity * p.PiecePrice * (1 – p.Discount)) * (1
– at.Discount)
      </POST>
    </OCL>
   </BehavioralLevel>
  </ProcessComponent>
...
        </Service>
```

Figure 4: Examples for the XML-based specification of components at behavioral level using OCL

In this example we use WCML-interfaces for prototypes on the syntactical level as shown in Figure 5. The WCML markup uses an official Document Type Definition that describes the grammar of the WCML components. For reasons of simplicity we do not use schemas and namespaces in this example (even though the WCML-compiler supports these). The WCML-component may be used as prototype for other components and for syntax checking in the WCML-Compiler.

```
<?xml version=”1.0" encoding=”UTF-8" ?>
<!DOCTYPE wcml SYSTEM “http://webengineering.org/REC/
wcml/wcml2.dtd”>
<wcml>
 <component uuid=”Order” referable=”false”>
   <property name=”TechnicallyPracticable” mode=”interface”
type=”boolean” />
    <property name=”Delivered”
 mode=”interface” type=”boolean” />
    <property name=”InvoiceAmount”
 mode=”interface” type=”double” />
     <property name=”Discount”
 mode=”interface” type=”double” />
    ...
 </component>
        </wcml>
```

Figure 5: The specification of a component at syntactical level using WCML-prototypes

3 CONCLUSION

We have pointed out that the coarse-grained implementation model of the Web hinders the representation of abstract design concepts in actual code. The resulting gap between implementation and design model is a burden to the use of modern software engineering practices in Web projects. The WebComposition approach with its implementation technology WCML bridges this gap and allows designing for reuse by a dedicated process model.

The WebComposition Process Model describes a consistent approach to the development of Web-applications as component software. It introduces the concept of an evolution-oriented process model that allows for the integration of components using the abstract concept service. In this view a Web-application is a set of services that are grouped by certain domains. The services are modeled as components with the WebComposition Markup Language. WCML is an application of XML and is in concordance with the basic principles of the Web. The application domains are described through services that correspond to domain-components. The evolution by application domains is a central part of the process model and is described through the so-called Evolutionbus, a framework for the integration of domain-components. In this paper we proposed a standardized way to describe components on a software contract level architecture. The use of a standardized approach like presented in this paper is essential for a disciplined evolution of Web-applications.

The WebComposition Process Model has been successfully applied to several real world applications. The advantages for the evolution could be verified in a three-year project for a large international Web-application “E-Victor Procurement Portal” at the company Hewlett-Packard that has been developed according to the process model, and using the described component technologies.

EXAMPLES

For information about the WCML-Compiler and the WebComposition approach please feel free to browse: http://webengineering.org

REFERENCES

Ehrig, H., & Mahr, B. (1985). *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Berlin: Springer.

Gaedke, M. (2000). *Komponententechnik für Entwicklung und Evolution von Anwendungen im World Wide Web*. Aachen: Shaker Verlag.

Gaedke, M., Schempf, D., & Gellersen, H.-W. (1999, May 11-14, 1999). *WCML: An enabling technology for the reuse in object-oriented Web Engineering.* Paper presented at the Poster-Proceedings of the 8th International World Wide Web Conference (WWW8), Toronto, Ontario, Canada.

Gellersen, H.-W., & Gaedke, M. (1999). Object-Oriented Web Application Development. *IEEE Internet Computing, 3*(1), 60-68.

Meyer, B. (1988). *Object-Oriented Software Construction*. Englewood Cliffs: Prentice Hall.

Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, & Softeam. (1997). *Object Constraint Language Specification: Version 1.1, 1 September 1997*. Available: http://www.rational.com/uml [1999, 04-17].

Turowski, K. (1999). *Standardisierung von Fachkomponenten: Spezifikation und Objekte der Standardisierung*. Paper presented at the 3. Meistersingertreffen, Schloss Thurnau.