

A Componentware Methodology based on Process Patterns

Klaus Bergner, Andreas Rausch
Marc Sihling, Alexander Vilbig

Institut für Informatik
Technische Universität München
D-80290 München
<http://www4.informatik.tu-muenchen.de>

30th April 1998

Abstract

We present a new approach to a componentware development methodology based on a system of process patterns. We argue that organizing the development process by means of a pattern system results in much higher flexibility and is especially suited to the context of componentware.

1 Introduction

Componentware is concerned with the development of software systems by using components as the essential building blocks. The most common understanding of a component is that of an encapsulated unit of software with well-documented and therefore well-understood interfaces. Via those interfaces, a component can be connected to other components. Systems built of loosely coupled components promise a higher degree of software reuse and improved handling of the involved complexity. Ideally, system development can thus be reduced to a simple composition of existing components and their adaptation to specific needs.

However, to leverage these benefits componentware must be supported by an appropriate development methodology based on a flexible process model. We argue that existing process models are not adequate to componentware. Instead, we propose to apply a result-oriented, pattern-driven process that can be adapted dynamically to the current state of the project. After outlining the structure of our methodology, we propose a first preliminary version of a componentware process pattern language.

2 Componentware Methodology

2.1 Requirements for a Componentware Methodology

Componentware inherently deals with reuse of existing software. As such, it does not fit very well with many of the existing methodologies tailored to developing software from scratch.

Traditional top-down approaches start with the initial customer requirements and refine them continually until the level of detail is sufficient to allow implementation. This involves some severe drawbacks: Initially the customer often does not know all relevant requirements, cannot state them adequately, or even states inconsistent requirements. Consequently, many delivered systems do not meet the customer's expectations. In addition, top-down development leads to systems that are very brittle with respect to changing requirements as the system architecture and the involved components are specifically adjusted to the initial set of requirements.

In a bottom-up approach, existing components are iteratively composed and combined into higher-level components until, finally, a top-level component emerges which fulfills the customer's requirements. Obviously such an approach is impractical in most cases because the requirements are not taken into account early enough.

Hence, practical experience has shown that neither a pure top-down nor a pure bottom-up approach are adequate in the context of componentware. Cyclic process models try to overcome the deficiencies of both approaches by structuring the whole process as a series of iterated short phases which are typically organized in a waterfall fashion themselves. We feel that even this approach is too rigid in the context

of componentware. Taken seriously, a project manager may only react to changes of the current situation by starting a complete development cycle—otherwise, he violates the cyclic nature of the process model. This is simply not adequate in many situations, for example when new components become available during development or when existing systems are to be reengineered. We think that a flexible process should be much more modular and tailor-made to the current state of the project, much in analogy to the essential properties of the components themselves and the systems they are a part of.

Our basic idea is to use a result-driven approach in which structured and well-defined units of development information are the products of the various development activities and serve as a clear interface between them. Currently, we do not further define the exact nature of these development information units although in most cases they comprise common development documents like requirements specifications, user manuals or system design specifications. Based on this result structure and inherent consistency criteria, the actual sequence of activities—the process—may be composed very flexibly.

The result structure is comparable to a filing cabinet with many drawers for the development documents. During the corresponding activities of the software development process, these drawers are continually filled. Despite its inherent flexibility, such a process is quite easy to control. The actual extent to which the drawers are filled serves as a valuable indicator for the progress of the project. Compared with the time-based progress metrics of conventional linear methods this is certainly an advantage. It provides even more detailed and meaningful information about the project than the number of finished cycles that is usually taken as a progress indicator in a cyclic model.

2.2 Result and Activity Structure of the Methodology

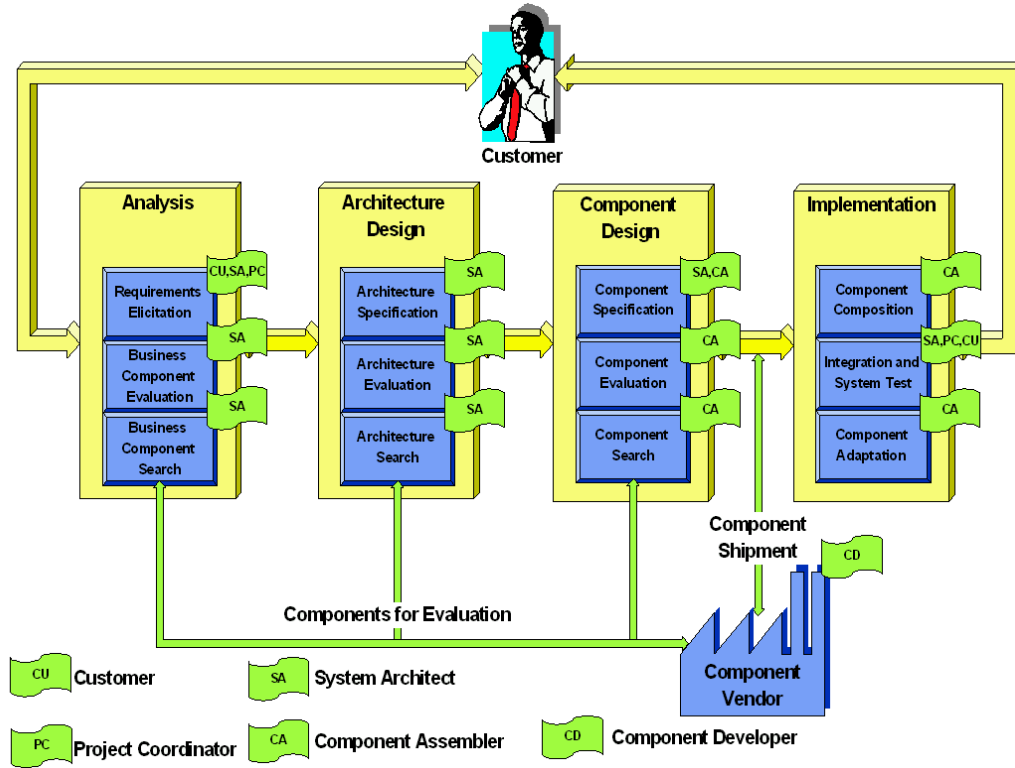


Figure 1: Result and Activity Structure

Figure 1 illustrates our proposal for a component-based development model. The main activities are similar to the phases of the conventional waterfall model: Analysis, System Design, Component Design, and Implementation. As mentioned above, we do not require these activities to be performed sequentially, but allow a flexible process that can be tailored using development process patterns. The flow of structured development information serving as an interface between the activities is represented as bold three-dimensional arrows in the figure. For example, the interface between the activities Analysis and Architecture Design might comprise a requirements analysis document, a user manual, and an administrator manual.

Each main activity consists of several sub-activities. Although these sub-activities may be carried out in parallel and do influence each other, there may not necessarily exist well-defined interface documents between them. Therefore the corresponding boxes—for example, Requirements Elicitation, Business-Oriented Component Evaluation, and Business-Oriented Component Search within the main activity Analysis—are represented without arrows between them in Figure 1.

The figure also includes some special activities concerning the interaction with component vendors. The sub-activity Component Search within Component Design, for example, requires a search in several repositories provided by component vendors. Some of the components found by this search will be subject to an evaluation which determines if they match the requirements and fit into the existing architecture. Whenever components are delivered by vendors, these components have to be tested against the specification and the corresponding test scenarios.

Note that the different activities are performed by individuals or groups in different roles. System architects, for example, develop the overall architecture of a system and supervise the development process, whereas component assemblers are responsible for the adaptation and composition of existing components.

Due to the space restrictions of this position paper, we do not describe the activities and roles of the proposed methodology in more detail .

3 A Result-Based Development Process Pattern Language

According to most authors, patterns consist of three main parts: The *context* is an overall situation giving rise to a certain recurring *problem* that may be solved by a general and proven *solution* [BMR⁺96].

Within our approach, these three parts may be described as follows:

- The *context* is given by the current state of the development project, characterized by the state and consistency of the development documents and the role structure of the development team. The current state of the development documents, however, has a stronger influence on the next development steps. Using the cabinet-drawer metaphor, the current state can be easily visualized using a variant of Figure 1.
- The *problem* describes a concrete problem that typically arises in the given context. It mentions the external forces, namely, the influences of customers, competitors, and component vendors, the time and money constraints, as well as the requirements and desirable properties the solution should have. Only by balancing all of these forces, the overall development process and its individual activities can be planned adequately.
- The *solution* comprises the involved roles and development activities and provides a concrete recommendation for the overall process or the next actions.

Following this approach, a process may be designed on multiple levels of detail—ranging from very fine-grained patterns, giving advice on single activities to high-level patterns for the design of the overall development process. Like in other pattern languages, the single patterns may also be combined with each other, forming a multi-level system of different patterns. In the following sections, we present a first proposal for such a pattern system consisting of four different levels.

There is already an established literature about process and organizational patterns. The most relevant contribution seems to have been made by Coplien who proposes a generative pattern language to describe existing and construct new organisations [Cop94]. He considers rather general patterns that mostly cover the organizational and social aspects of the software development process, referring to problems like the ideal size of an organization or the right communication structure. These patterns are not closely tied to an underlying methodology, claiming that a good set of organizational patterns helps to indirectly generate the right process. We feel that both methodology and organizational aspects are of equal importance to successful software development. However, it should be possible to use the organizational patterns in conjunction with our proposed development process patterns.

3.1 Overall Patterns

Overall patterns pertain to the overall development approach chosen. They are concerned with the logical and temporal relations between the four main development activities Analysis, Architecture Design, Component Design, and Component Implementation.

Currently, we have identified the following overall patterns:

Waterfall (alias Sequential) corresponds to the traditional, linear sequence of the main development activities, starting with Analysis and ending with Implementation. Waterfall can be combined with local cycles to yield an enhanced waterfall pattern.

Spiral (alias Cycle) corresponds to Boehm's spiral model; the activities are traversed in a cyclic fashion, starting with Analysis, and ending with Implementation.

Reverse Waterfall (alias Bottom-Up) is a typical reengineering process. Based on an existing implementation, design and analysis documents are built.

Roundtrip corresponds to Booch's roundtrip process—starting with Analysis and progressing to Implementation, then reversing the process to ensure consistency of design and analysis documents with the implementation.

Inside-Out (alias Architecture-Driven) describes a process which is based on a given system architecture. Starting with this architecture, the developer tries to elicit reasonable user requirements and builds an implementation. This is a very common approach, especially in the context of componentware.

Outside-In may be applied in projects where neither user requirements nor the technical issues are clear at all. The developer begins with collecting the requirements and implementing an experimental prototype at the same time, hoping to reach a suitable system architecture afterwards.

3.2 Cross-Activity Patterns

Cross-activity patterns are patterns pertaining to the interrelationships and the document consistency between the main activities. We propose the following, yet preliminary examples for such patterns:

Development Status is applied to get a clear view on the status of the current results and documents.

Analyse Impacts of Change has to be performed whenever it is not clear which documents must be updated in reaction to certain changes. A special case is **React on New Component Version**.

Consolidate Documents corresponds to achieving consistency between the documents of different activities.

Experimental Prototype corresponds to building a preliminary version of the full system to get familiar with the technical foundations. The experience gained has to be transferred into the system architecture.

Explorative Prototype corresponds to building a preliminary version of the full system to analyze the user's requirements. The experience gained has to be transferred to the user requirements document.

Formal Verification can be done based on requirements or technical criteria, if the correctness of a certain component is of utmost importance.

Make-or-Buy corresponds to the decision whether a component is to be produced in-house or bought from an external vendor. This pattern belongs to the cross-activity category because information from all main development activities is usually necessary to reach this decision.

Change Architecture is done by decomposing a component system into its single components and re-composing them according to a new overall design. A change of the architecture usually has serious impacts on the implementation and on the way requirements are fulfilled.

3.3 Intra-Activity Patterns

Intra-Activity Patterns are concerned with the relation between the development tasks belonging to a given main activity. Some examples are:

Requirements-Driven Analysis pertains to the way the activity Analysis is organized. In this case the criteria of the sub-activity Component Evaluation are derived from the requirements provided by the client.

Component-Driven Requirements Elicitation is the complementary approach. In this case the component evaluation is driven by the features of existing components. The best of them are selected and proposed to the client as a possible solution. Component-Driven Requirements Elicitation is adequate when the customer has yet no clear understanding of the requirements and is willing to standardize his business processes according to an existing solution.

Analogous alternatives exist for the other main development activities.

3.4 Local Patterns

Local patterns apply to single sub-activities. For technical sub-activities like Component Composition or Component Adaptation, this corresponds to design patterns similar to those developed for object-oriented software systems. Obviously, componentware requires special patterns: Adaptation patterns, for example, deal with process of adaptating existing components via wrappers, inheritance, or even source code rewriting. Composition patterns describe possibilities for the composition of existing components. These patterns closely resemble the structural patterns of [GHJV95].

Other local patterns are involved with sub-activities like Component and Architecture Search, which could be realized by performing a market study, for example. Test and integration patterns pertain, among others, to white-box testing, black-box testing, code reviews, and so on. Due to the large number of conceivable local patterns, we currently do not attempt to propose an elaborate and sufficiently complete pattern catalog.

4 Conclusion

We have presented a first, preliminary version of a methodology and a pattern language for organizing the development process of component-oriented systems based on a given, standardized result structure. We believe that this approach allows for the flexibility and modularity needed in real-world development. Currently, the proposed process model and its accompanying pattern language are far from being complete—both structure and content of the pattern catalog are not sufficiently elaborated. Furthermore, the individual patterns themselves need to be formulated in more detail. These areas are the focus of our future work.

References

- [BMR⁺96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. Wiley & Sons, 1996.
- [Cop94] J. O. Coplien. A development process generative pattern language. In *PLoP '94 Conference on Pattern Languages of Programming*, 1994.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.