A Componentware Development 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

29th July 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 higher flexibility compared to traditional ways of defining development processes. This is especially important in the context of componentware. Finally, we propose a pattern catalog with a selection of suitable process patterns.

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 inherently involved complexity. Ideally, system development can thus be reduced to a simple composition of existing components and their adaptation to specific needs.

To leverage the technical advantages of componentware and to support the reuse of existing components, the introduction of new roles, tasks, and results in the process model is immanent. This leads, for example, to the separation of the roles *Component Developer* and *Component Assembler* as well as to new development tasks like searching and evaluating existing components (cf. Section 3).

Obviously, the introduction of new roles and tasks requires a process model tailored to componentware. However, this doesn't imply that the process model in question has to be completely new and revolutionary. After all, componentware itself is an evolutionary approach based on the foundations of earlier paradigms like object-orientation. Therefore, a proposed componentware methodology should represent an adapted and improved version of established methodologies. We believe that a pattern-based approach suits especially well, as patterns inherently try to capture proven solutions and established practice.

The use of patterns also allows for additional flexibility because neither the context nor the proposed solution is defined formally. This is particularly suited to the situation in project management, as formal definitions and hard rules are not adequate in this area. Accordingly, the rigidity of prescriptive process models is widely felt as a strong drawback and there is common agreement about the need to adapt the process to the actual needs. Jacobson, for example, talks about the future UML process as a "strawman" [Jac98] serving only for explanatory purposes and probably never applied exactly as proposed. Henderson-Sellers states in [HS96] that "a method has NO ROLE as a recipe book by which a series of steps is followed slavishly". In our opinion, statements like these and common practice clearly indicate deficiencies in traditional process models or, at least, in the sequential and prescriptive form in which they are presented.

In our impression, the key requirement for a componentware process model is the efficient combination of top-down and bottom-up development. This results in systems which fulfill the customer's requirements while reusing existing high-quality components. Cyclic and iterative process models try to merge the advantages of top-down and bottom-up approaches by structuring the whole process as a series of iterated short phases. These phases are typically organized in a top-down fashion themselves. Thereby experience gained at the end of a given cycle may influence the next iteration in a bottom-up way. However, we feel that even cyclic approaches are 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, the cyclic nature of the process model is violated. This criticism also applies to other prescriptive approaches to componentware, like the reuse-driven development process presented in [Sam97]. Although the suggested sequence of tasks seems quite reasonable, it is obviously not adequate for all componentware projects or project situations.

In this paper, we propose a pattern-oriented process model that may dynamically be adapted to the current situation of the project. After outlining the basic concepts in Section 2, we present its main constituents, namely roles (Section 3), result structure (Section 4), and a catalog of process patterns (Section 5). After covering related approaches in Section 6, a short conclusion ends the paper. The appendix contains a number of exemplary process patterns which illustrate our approach.

2 A Pattern-Based Process Model

The conceptual framework of the presented pattern-based process model consists of three main building blocks:

- A well-defined hierarchical *result structure* with units of development information representing results of the various development activities.
- A set of *consistency criteria* which apply to the different units of development information and the relationships between them.
- A set of *process patterns* providing guidelines about the organisation of the actual development process. A given pattern suggests a sequence of development activities which are advisable in the current situation of the project. The state and the consistency of the result structure constitute essential parts of this context.

Result Structure: The result structure is comparable to a filing cabinet with drawers for the different development documents. In the following chapters we present a particular result structure which is suited to the development of component-based systems. Because the actual results and their structure depend on the nature of the development project in question, we distinguish two ways of customizing the result structure. While *tailoring* allows to simply exclude irrelevant results, *adaptation* also comprises the introduction of additional results. Consequently, in the latter case, modified or even new process patterns have to be introduced as well. Note, however, that we do not allow for the modification of the result structure during the course of a project which seems to be an acceptable restriction in practice.

Once a development project has started, the result structure is continually "filled" by individuals or groups performing specific development actions. The crucial point is that we do *not* prescribe the sequence of the individual actions a priori—instead, the actual process will be designed and continually modified by the project manager according to a catalog of process patterns.

Consistency Criteria: Various consistency criteria applying to the results of the development process may be derived from the syntactical and semantical relationships between them. An example of such a relationship between units of development information is *refinement*. It expresses the concept that a document contains additional, more concrete—albeit not conflicting—information compared to another, more abstract document.

We distinguish between certain parts of a development document which are related to other documents, and other parts which are considered to contain strictly internal information. Only the former parts are subject to the application of consistency criteria and may therefore be regarded as the "interface" of this document. For example, a Component Repository Document may contain a detailed report about existing components on the market. From the large variety of covered aspects, however, the *System Architect* is only concerned with the technical characteristics of particular, selected components.

Process Patterns: 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 parts may be described as follows:

- The *context* is defined by the current internal state of the development project and its external state with respect to the customers, the competitors, and the market situation. The internal state is characterized by the state and consistency of the result structure. The context is subject to the sections **Applicability** and **Structure** of the proposed pattern description format as shown in Section A of the appendix.
- The *problem* describes a concrete problem that typically arises in the given development context. It mentions internal or external forces, namely, the influences of developers, customers, competitors, component vendors, the time and money constraints, and the requirements and desirable properties the solution should have. A problem reflects an unbalance between these involved forces and therefore implies a call for action. We address a problem description in the sections **Intent** and **Motivation** of the description format. Note, however, that the theoretical distinction between problem and context is often not very clear in practice.

- The *solution* comprises the involved roles and parts of the result structure and provides concrete recommendations for an appropriate sequence and priority of development actions. It is represented in the sections **Structure**, **Tasks and Roles**, and **Application Guidelines** (cf. Section A).

The application of process patterns may be outlined as follows: Based on the project's current situation as partly represented by the state and consistency of the result structure, the *Project Coordinator* tries to identify current development problems. This information leads to the selection of one or more process patterns which may be applied because their context and problem descriptions match the current situation. After a careful consideration of the implied consequences, a pattern is chosen which recommends a number of development actions, their sequence and priority. An example for such a process pattern is **Explorative Prototyping**[1] (cf. Section B.3), stating under which preconditions it is advisable to build an explorative prototype, and the way this may be accomplished.

Despite its inherent flexibility, a process constructed by applying process patterns is quite easy to control because the consistency of the result structure and the actual extent to which it is elaborated serve as valuable indicators for the progress of the project. This way, even more detailled and meaningful information about the project is provided compared to the number of finished cycles which is usually taken as a progress indicator in a cyclic process model.

Explorative Prototyping as well as other process patterns are presented in the pattern catalog in Section 5. They have familiar names and are obviously not revolutionary. This is in accordance with the inherent principles behind all pattern-based approaches. Patterns are used to capture well-known knowledge and proven solutions in a given area of work. However, most of the process patterns described in this paper stem from older paradigms and have not yet been tested in the context of componentware. Although this violates in some sense the principle of a "proven solution", we believe that the evolutionary nature of the componentware approach itself permits the use and adaptation of process techniques from other development paradigms.

At present, we have no experience with the use of our process patterns in an industrial project—the practical evaluation of our approach will be the next step after consolidation of the pattern catalog. The current patterns try to capture the project management techniques applied during numerous practical project courses that have been conducted over the last five years [Tec94, Tec95, BBHP95, Ber96, Tec96, Tec97, BH97, Tec98b, Tec98a].

3 Roles

Componentware introduces a number of new roles for developers and project managers. Particularly noticeable are the separation of *Component Developer* from *Component Assembler*, and the introduction of a dedicated *System Architect* and a *Project Coordinator*. Other traditional roles, like *System Analyst*, will probably persist, although their focus and responsibilities will change. In our proposed methodology, the different developer and manager roles are defined as follows:

Component Developer: Components are developed by specialized component vendors or by in-house reuse centers as a part of large enterprises. The responsabilities of a *Component Developer* are to recognize the common requirements of many customers or users and to construct reusable components accordingly. If a customer requests a component, the *Component Developer* offers a tender and sells the component.

Component Assembler: Usually, complicated components have to be adapted to match their intended usage. The *Component Assembler* adapts and customizes pre-built standard components and integrates them into the system under development.

System Analyst: As in other methodologies, a *System Analyst* elicits the requirements of the customer. Concerning componentware, he also has to be aware of the characteristics and features of existing systems and business-relevant components.

System Architect: The *System Architect* develops a construction plan and selects adequate components as well as suitable *Component Developers* and *Component Assemblers*. During the construction of the system, the *System Architect* supervises and reviews the technical aspects and monitors the consistency and quality of the results.

Project Coordinator: A *Project Coordinator* as an individual is usually only part of very large projects. He supervises the whole development process, especially with respect to its schedule and costs. The *Project Coordinator* is responsible to the customer for meeting the deadline and the cost limit.

4 Result Structure

Figure 1 illustrates our proposal for the organization of the result structure underlying the process model. The main parts are resembling the phases of conventional process models although we explicitly separate business-oriented design

[1] For better understanding the names of patterns are printed in **bold font**, the names of roles are printed in *italic font*, and the names of development results are printed in serifless font.

from technical design: Analysis, Business Design, Technical Design, Specification, and Implementation. All main development results, like Business Design, consist of subresults like Architecture Design, Component Design, Evaluation, and Search. The consistency conditions between certain parts of the main results are visualized by arrows.

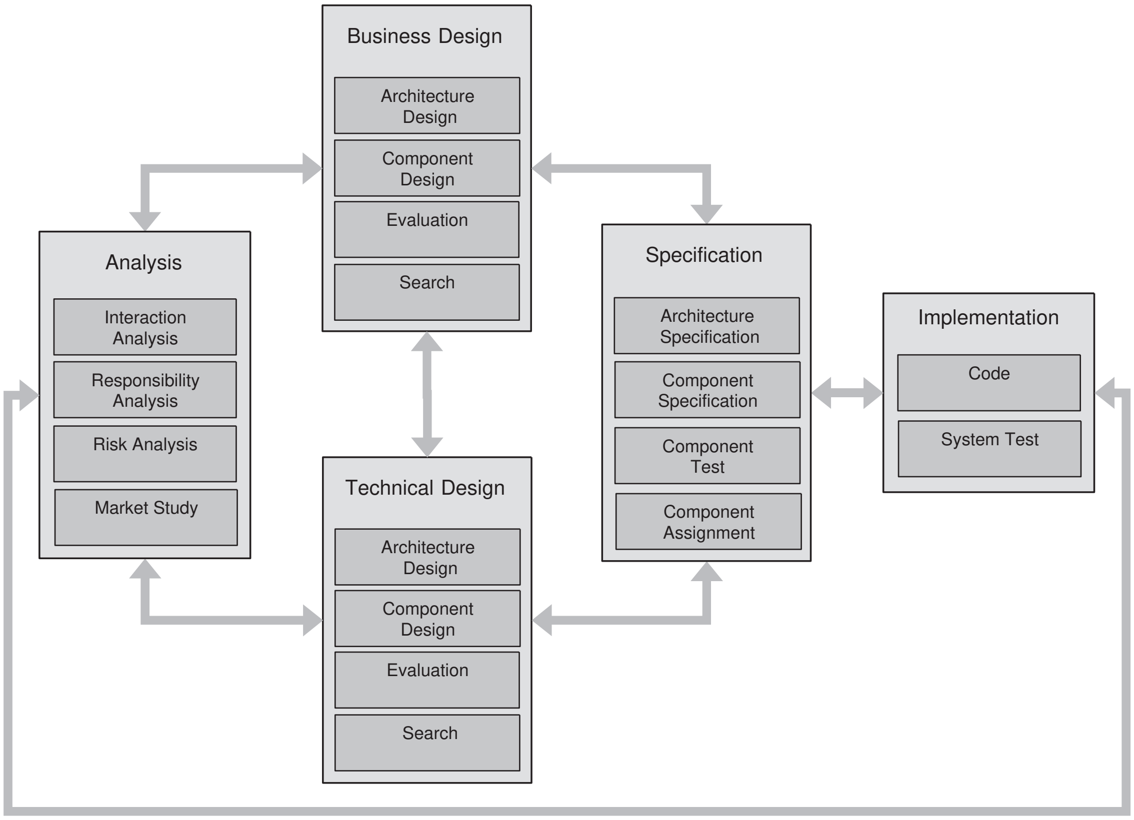

Figure 1: Result Structure

Note that there are no arrows between the subresults in a main result. This is due to the fact that these subresults are even closer coupled than the main results and are usually developed together. As componentware is based on reusing existing software, it is not plausible, for example, to design the technical architecture (subresult Architecture Design of main result Technical Design) without searching for and evaluating existing technical components (subresults Search and Evaluation of the same main result). In the following sections we describe the result structure in more detail.

4.1 Analysis

The Analysis main result contains the specification of the customer requirements.

The subresult Interaction Analysis is concerned with the interaction between the system and its environment. It determines the boundary of the system, the relevant actors (both human and technical systems), and their usage of the system to be developed. Contained may be parts like an overall Use Case Specification, a Business Process Model, Interaction Specifications including System Test Cases, and an explorative GUI Prototype.

The subresult Responsibility Analysis, on the other hand, specifies the expected functionality of the system with respect to the functional and non-functional user requirements. It describes the required services and use cases of the system in a declarative way by stating *what* is expected without prescribing *how* this is accomplished. Contained are parts like Service Specifications, Class Diagrams, and a Data Dictionary.

The subresult Risk Analysis identifies and assesses the benefits and risks associated with the development of the system under consideration. In the context of componentware, this requires a Market Study with information about existing business-oriented solutions, systems, and components.

Note that Analysis usually not only covers functional and non-functional requirements, but also technical requirements restricting the technical architecture of the system to be built. While the functional requirements must be fulfilled by the Business Design main result, the non-functional and technical requirements must be compliant with the Technical Design main result. Furthermore, the Implementation must pass the System Test Cases.

4.2 Business Design

Business Design defines the overall business-oriented architecture of the resulting system and specifies the employed business components.

The subresults Architecture Design and Component Design are comparable to the Interaction Analysis and Responsibility Analysis subresults of Analysis. However, they do not address technical issues, but instead provide a detailed specification of the business-relevant aspects, interactions, algorithms, and responsibilities of the system and its components. Search corresponds to a preselection of potentially suiting business components and standard business architectures that are subject to a final selection within the Specification main result. Within Evaluation, the characteristics of the found components and architectures are balanced against the criteria identified in Architecture Design and Component Design.

4.3 Technical Design

Technical Design comprises the specification of technical components, like database components, for example, and their overall connection architecture which together are suited to fulfill the customer's non-functional requirements. As this result deals with technical aspects of the system like persistence, distribution, and communication schemes, Technical Design represents a dedicated part of the development results that should be logically separated from Business Design.

In the context of componentware, however, the applied development principles are the same for both areas. Consequently, the involved subresults are analogous to those of Business Design.

4.4 Specification

The main results Business Design and Technical Design are concerned with two fundamentally different views on the developed system. The Specification main result merges and refines both views, thereby resulting in complete and consistent Architecture and Component Specifications.

As said above, both Business Design and Technical Design cover an evaluation of existing components from the business and technical point of view, resulting in a preselection of potentially suitable components for the system. The Specification subresult Component Test contains the results and test logs of these components with respect to the user requirements and the chosen system architecture.

Some of the desired components may simply be ordered whereas other components are not available at all and must be developed. The Component Assignment subresult specifies which components are to be developed in the current project and which components are ordered from external component suppliers or in-house profit centers. If a component is to be developed outside of the current project, a new, separate result structure has to be set up. Note the close correspondence between Architecture Specification and Component Specification on the one hand, and Interaction Analysis and Responsibility Analysis on the other hand. It allows for a clear hand-over of a component specification to a component developer outside the project.

4.5 Implementation

The most important subresult of the Implementation main result is of course the Code of the system under consideration. It comprises source code as well as binary-only components. The other subresult covers the System Test results.

5 A Process Pattern Language

In our approach, a process may be designed on multiple levels of detail, ranging from very fine-grained patterns, providing advice for the creation of a single subresult, to high-level patterns for the organisation of the overall development process. Like in other pattern-based approaches, the single patterns may also be combined with each other, forming a multi-level system of patterns. In the following sections, we present a proposal for such a pattern system at four different levels: project patterns, inter-result patterns, main result patterns, and subresult patterns.

We use the format explained in Section A of the appendix to describe the individual patterns. The format is based on the template in [GHJV95] which should be familiar to most readers and allows to structure the information in a concise and uniform way. Each pattern description is preceded by the pattern name which conveys the essence of the pattern. Selected process patterns are described in detail in Section B of the appendix whereas a short form is used in the following sections to provide an overview of the proposed pattern catalog.

5.1 Project Patterns

Project patterns apply to the overall development approach chosen. They are concerned with the logical and temporal relations between the creation of the main results Analysis, Business Design, Technical Design, Specification, and

Implementation:

▷ **Top-Down (alias Sequential, Waterfall):** Create the main results sequentially, starting with Analysis, continuing with Design and Specification, and ending with Implementation.

In some projects, the duration and effort of the development activities may be planned very well. An example is a development project for a host-based bank accounting system in which the user requirements are well-known and there are no particular technical and organisational risks involved.
The project starts with the analysis of the complete set of user requirements by the *System Analyst*. Based on this result, the *System Architect* develops matching Business Design and Technical Design main results. The Specification is outlined by the *System Architect* and further elaborated by the *Component Assembler*, who subsequently implements and tests the system against the customer requirements.
In a **Top-Down** process, work on a new main result may begin only after the previous main results have been finished. If errors are discovered, it is possible to redo already finished results. However, in this case all dependent results must be reworked, too. The main advantage of applying this pattern is a manageable and traceable development process with clear milestones after each main result phase. The main risks involve unforeseen technical or organisational issues and unrecognized or modified customer requirements which may remain undiscovered over long periods of time, leading to a costly reworking of already elaborated results. There may also be negative effects on the motivation of the developers because a runnable system is becoming apparent only at the very end of the process. ◁

▷ **Bottom-Up (alias Reverse Waterfall):** Starting out from proven solutions and existing components, elaborate a matching design and reasonable requirements in a linear process.

Such an overall process applies to situations in which inital user requirements are only roughly stated, but well-known standard solutions and many reusable components exist. As an example, consider the development of a web-based database interface. The particular query features are not specified in detail by the customer but are rather supposed to be "as usual".
The *System Architect* together with the *Component Assembler* start out from well-known solutions—most likely involving standard components—and together elaborate or select a reasonable Design of the system. Existing implementations may be analyzed to gather useful development information. At the end of this process, the *System Analyst* creates the documents Interaction Analysis and Responsibility Analysis of Analysis in compliance with the features of the built system.
After finishing the process, the customer may now refine his requirements, probably necessitating a rebuild of the system supported by a different project pattern like **Top-Down** or **Iterative**. A **Bottom-Up** process is particularly suited to projects with tight time constraints, as only few efforts have to be spent for analysis, and the system is largely built from existing components. ◁

▷ **Architecture-Driven:** After creating the main Design results, elicitate reasonable user requirements and specify and implement the system.
⟶ *See appendix, Section B.1.* ◁

▷ **Roundtrip:** Start working on the Analysis main result and progress from Business Design, Technical Design and Specification to Implementation, then reverse the sequence to ensure consistency of Specification, Design, and Analysis results with Implementation. Repeat as necessary.

The **Roundtrip** process may be understood as a combination of interleaving **Top-Down** and **Bottom-Up** processes; it corresponds to a variant of Booch's roundtrip approach as introduced in [Boo94]. The main motivation for this pattern is to remove inconsistencies in the result structure which may arise, for example, in a **Top-Down** process when changes in the Implementation are not documented in the Design documents.
The **Roundtrip** approach is especially suited if a consistent and complete result structure is important for the further development and maintenance of a system in the long run. However, the pattern does also encourage a certain amount of temporary inconsistencies, allowing the developers to experiment with new variants and to diverge from specified solutions. ◁

▷ **Iterative (alias Cyclic, Incremental, Evolutionary Prototyping):** Create the main results in a cyclic fashion by repeated application of the **Top-Down** pattern. In every iteration, extend and adapt the result structure to consider additional or modified requirements.

Oftentimes, requirements are extended, modified or even redefined during the course of a project. An iterative process

copes with this uncertainty by restarting the development process after implementing a partial version of the system, called an "evolutionary prototype". As this prototype is not discarded, but used as the basis for the final system, its corresponding development documents are part of the original project's result structure.
Although the length of one iteration cycle depends mainly on the size of the increment to be implemented, it should not be too long—typical values are some weeks to few months. Short increments have positive effects on the motivation of the developers and the resulting evolutionary prototypes may even be marketed as preliminary product versions if time-to-market is important.
The main risk with an iterative approach is that new requirements may invalidate the evolutionary prototype's architecture. If this is the case, the effort needed for the conversion to a new architecture may be considerably high. ◁

5.2 Inter-Result Patterns

An Inter-Result Pattern suggests a solution which produces subresults in at least two different main results.

▷ **Tune Performance:** Analyze an occurring performance problem. Either redesign the overall Business Design and Technical Architecture or change the system Specification and Implementation.

If a performance problem occurs during the creation of the System Test subresult, the *System Architect* and *Component Assembler* analyze the problem and decide whether a local code optimization (without violating the Specification results) or a reworking of the overall design is required to solve the problem.
In the former case the *Component Assembler* optimizes Code, Architecture Specification, and Component Specification individually. Considerable care has to be taken that these subresults remain consistent. If tests show that the optimization does not reach the expected performance, it may be necessary to redesign the architecture.
In case of an architectural redesign the *System Architect* reconsiders both technical and business Architecture Design as well as Component Design. During this process he has to ensure that the resulting integrated Architecture Specification and Component Specification yield the expected performance gain. If this cannot be reached, business design and technical design are probably not compatible, and the overall architecture has to be questioned. ◁

▷ **Re-Engineering:** Analyze an existing system in order to reuse and improve as many parts as possible.

Many development projects do not implement a system from scratch but integrate, extend, or partly replace existing systems. This may necessitate an in-depth evaluation of these systems, leading to an extensive amount of development information. In order to integrate this information into the current project, additional Current State Analysis and Current State Evaluation documents within several subresults of all main results are produced.
The *System Analyst* starts with creating these current state results within the Analysis main result in order to figure out obsolete, changed and new user requirements. After this task is partially completed, the *System Architect* decides which other areas of the result structure are concerned and which components of the existing systems should be examined. Most of the involved results may then be elaborated in parallel.
After the current state analysis and evaluation have been completed, both *System Architect* and *Component Assembler* subsequently transform the remaining (and therefore still valid) results in the Current State Analysis and Current State Evaluation documents into "ordinary" results of the currently developed system.
An example for such an approach, consider a host-based legacy system which is to be re-engineered for a client/server environment. In this case, it may be vital to reuse the Business Design as far as possible while dismissing the Technical Design. In contrast, if a company developed a CASE tool on top of an object-oriented repository and currently plans to implement a CAD tool, it is the technical design which may probably be reused for the new project. ◁

▷ **Reverse Engineering:** Analyze existing code in order to understand and reuse as many aspects as possible of the existing desing and implementation.

Oftentimes during **Re-Engineering**, one has to consider design and architecture of legacy systems although their documentation is not available anymore. In this case, *Component Assembler* and *System Architect* first produce a Current State Analysis document for the subresult Code of the main result Implementation. The *System Architect* then recovers and figures out the Current State Analysis and Current State Evaluation documents of the appropriate subresults in Business Design and Technical Design. This process is usually supported by CASE tools and code analyzers.
Note that this pattern may also be applied to acquire information and know-how from competitors. ◁

▷ **Periodical Build:** Build a compilable or runnable version of the system to assess or ensure the consistency of and between Implementation and Specification.

In large projects with many sub-teams the consistency of interfaces is often problematic, especially if there is no adequate tool support. This is not only critical because it leads to unnecessary efforts for re-establishing the consistency later on, but also because inconsistencies in the result structure make it impossible for the project managers to measure the progress of the project.
The *Component Assembler* periodically links and compiles builds within the Code subresult of the Implementation main result. The time between such builds depends on the size of the project and the effort needed for the build, but should be short enough to prevent the emerging of larger inconsistencies (typically, builds are done daily to at most weekly). If the build does not succeed, the *Component Assembler* analyzes the reason and revises the Implementation and Specification main results. If the overall design is affected, the *System Architect* has to rework the Business Design and Technical Design main results as well.
The process for periodical builds of runnable systems is similar, with the additional benefit that the results of the build may be used by the *System Analyst* for **Explorative Prototyping** to gain feedback from the customer or to gather further requirements. Runnable system builds may also positively effect the motivation of the developers because the progress of the project is apparent at all times. ◁

▷ **Technical Foundation:** Design and implement the technical infrastructure with a small subset of the business functionality before completing the whole system.

Sometimes, a working technical infrastructure is essential in order to implement and test the business functionality of a large system. As an example, consider a large product planning system depending on various specialized services for persistency, transactions, and communication.
In order to realize the necessary technical foundation, the Technical Design has to be completely elaborated by the *System Architect*. Furthermore, the Analysis and Business Design main results should clearly define at least one typical business process. It should also be evident that the remaining business processes may be realized with the chosen Technical Design, either because the technical design is very general and powerful or because the the business functionality is homogeneous and well-understood in principle.
Based on the completed Technical Design and a small part of the Business Design, *System Architect* and *Component Assembler* elaborate a first version of the Specification and Implementation main results. If the Evaluation and Test subresults for this first version are positive, the remaining parts of Business Design are specified and implemented based on the technical infrastructure. ◁

▷ **Combined Experimental Prototyping:** Build a preliminary, throw-away version of the full system in order to evaluate the integration of business-oriented and technical design aspects.

For complex systems it is a critical question whether Business Design and Technical Design match with each other and can be combined to a consistent system Specification. An example is a system supporting mobile workers during the maintenance of aeroplanes—it is not only critical how to represent the business-relevant data and computations, but also how to distribute them technically onto the mobile computers of the workers. In the context of componentware, it is furthermore interesting whether the components found in the Search subresult are suitable and sufficient to implement the overall design.
Based on reasonably complete sections of Business Design and Technical Design, the *System Architect* creates simplified, but consistent Architecture Specification and Component Specification subresults of the overall Specification. If this attempt is successful, the *Component Assembler* may implement this specification and test the resulting prototype for its compliance with existing components as well as for certain critical properties.
As the resulting prototype is only intended to demonstrate certain, critical aspects of the full system, it is usually built using specialized prototyping tools and a very simplefied process, resulting in poor overall software and documentation quality. To prevent the reuse of Combined Experimental Prototype, it is archived in a recursive result structure of its own as a part of the Specification main result. ◁

▷ **Component Assessment:** Assess the impact of a new component on requirements, design, and implementation of the developed system.

During the course of a development project, it is likely that promising new components or updated versions of already used components become available. Based on the current state of the project, it is necessary to assess the impact of a potential integration of these components into the system.
Component Assembler and *System Architect* perform an extensive risk study. First, they include the new or updated components in the Search and Evaluation subresults of Business Design and Technical Design. Subsequently,

the subresults of Component Test and Component Specification have to be updated or amended as well. Based on these results, a Integration Risk Analysis document is produced as part of the Architecture Specification and Component Specification subresults. This document mentions the involved risks and costs as well as the potential benefits and improvements of the integration. Attention should be paid to the effects on the overall architecture, the expected performance of the system and potential inconsistencies with existing components.
Additional aspects considered by the *System Analyst* cover the influence on the user requirements stated in the Analysis documents. The resulting system is liable to offer new or improved functionality that has to be balanced against the user's needs. Based on the outcome of the Integration Risk Analysis, the *System Architect*, the *System Analyst*, and possibly the *Project Coordinator* decide on the integration of the new component. ◁

▷ **Component Update:** React to an updated version of an existing component which is used in the developed system.

Frequently, improved and extended versions of an existing component become available. Oftentimes, these updated components are of a higher quality (due to bug fixes or optimizations, for example) and may offer several new features. In order to decide if these improvements justify an integration of the updated component, it is advisable to apply a **Component Assessment** which evaluates the potential impacts on the system. If there are only minor changes, the *Component Assembler* integrates the new component version into the system as part of the Code subresult. It may be necessary to adapt the component's interface (see the adaptation subresult patterns) or to remove workarounds that are now obsolete due to the removal of bugs. After the integration of the component into the system, regression tests should be performed to ensure correct system behavior. In case of major changes that extend to the technical or business architecture, it is advisable to treat the updated component as a novel component and apply the pattern **Component Innovation**. ◁

▷ **Component Innovation:** React to a novel component which becomes available during the course of the development project.

From time to time, components with novel capabilities become available on the market during the course of the development project. Sometimes, these components could replace or substantially improve an already developed part of the system. However, it is likely that the integration of a new component leads to considerable changes of the current architecture and component specifications. In order to decide if the potential improvements justify an integration of this component into the system, it is advisable to apply a **Component Assessment** which evaluates the potential impacts on the system. If the produced Integration Risk Analysis indicates a net positive effect, the integration requires a number of critical operations and transformations on the existing results. In most cases, it involves drastic changes to parts of the already existing Specification and Implementation, leading to detailed tests of these parts by the *Component Assembler*. They are documented in the Component Test subresult of the Specification. If a complete redesign seems adequate and economic, the *System Architect* has to rework and evaluate the subresults of Business Design, Technical Design, and Specification. ◁

5.3 Main Result Patterns

Main result patterns are concerned with the way a given main development result is elaborated. They do not imply new consistency criteria as the individual subresults of a main result usually have to be produced in a strongly coupled process.

▷ **Customer-Driven Analysis:** Determine the requirements in the Analysis main result predominantly based on information from the customer.

Many developed systems are targeted at a single customer who expects support for certain parts of his business processes. It is therefore necessary to carefully analyze these business processes and the intended functionality of the system in close dialog with the customer. Hence, the *System Analyst* first elaborates the subresults of Interaction Analysis and Responsibility Analysis before Risk Analysis and Market Study. Only if the latter subresults indicate potential drawbacks or existing solutions that imply a modification of the original requirements, the former subresults are reconsidered accordingly. ◁

▷ **Market-Driven Analysis:** Determine the requirements in the Analysis main result predominantly based on the characteristics of existing components and systems.

Sometimes the customer has yet no clear understanding of his requirements or there exists no defined customer for a developed system because it is targeted at a larger market. In this case, the gathering of requirements begins with an extensive Market Study that provides an overview of existing systems and their features as well as generally desired features of new systems in this area. Based on this information, the *System Analyst* derives plausible requirements that are documented in the subresults Interaction Analysis and Responsibility Analysis. Once tentative requirements have been elaborated, it is necessary to evaluate the involved development risks as part of the Risk Analysis subresult. In the case of a market-driven analysis, it is especially important to consider the predicted future development of the market and the strategy of competitors. Note that in case of an existing customer, it is possible to apply **Explorative Prototyping** based on existing components to help the customer discover and clearly state the system's requirements. ◁

▷ **Experimental Prototyping:** Build a preliminary throw-away version of the full system or important parts to get familiar with the business or technical architecture.
⟶ *See appendix, Section B.2.* ◁

▷ **Explorative Prototyping:** Build a preliminary throw-away version of the system or some of its vital components in order to explore and analyze the customer requirements.
⟶ *See appendix, Section B.3.* ◁

▷ **Design-Driven Evaluation:** Derive the criteria for the Evaluation of existing, reusable components and architectures within Business Design and Technical Design main results from already created Architecture Design and Component Design subresults.

A comprehensive Evaluation of existing components and architectures for Business Design and Technical Design is a critical step in the development of component-oriented systems. The relevant evaluation criteria may be derived in a top-down approach from already present results of Architecture and Component Design by the *System Architect*. This ensures compliance with the already specified or developed components and the chosen architecture. Such an approach to evaluation is advisable if widely used or even standardized components and architectures (both technical and business-oriented) dominate system development in the given area. In this case it is conceivable that there exist standard evaluation criteria and conformance test procedures which may be applied. ◁

▷ **Component-Driven Evaluation:** Derive the criteria for the Evaluation of existing, reusable components and architectures within Business Design and Technical Design main results from the features of these components, the best of which are selected and combined in the design.

An inverse approach to Evaluation in Business Design and Technical Design (as compared to **Design-Driven Evaluation** is to gather comprehensive evaluation criteria from an analysis of the existing components and architectures as stated in the created Search subresult. It is likely that many of the discovered components and architectures in the given application domain share a set of core features and fulfill a set of common requirements. These aspects of existing solution are a good starting point for their evaluation by the *System Architect*, especially if there are yet only few results in Architecture and Component Design. Once suitable components and architectures have been preselected as a result of Evaluation, it is possible to elaborate Architecture and Component Design and apply the pattern **Design-Driven Evaluation** to further refine the Evaluation subresult. ◁

5.4 Subresult Patterns

Subresult patterns apply to the creation of single subresults. For technical subresults like Architecture Specification or Component Specification, this leads to specialized componentware design patterns similar to those developed for object-oriented software systems. Obviously, componentware requires certain specific variants. Adaptation patterns, for example, deal with the process of adapting existing components by means of wrappers, inheritance, or even source code modification. Two examples of these rather technical patterns which concern the subresult Code of the main result Implementation are given below:

▷ **Adaptation by Wrapping:** Adapt the behavior of an existing component by embedding it in a "wrapping" component.

Sometimes, an existing business or technical component has to be adapted in order to be reused in the current development project. Although many components provide a predefined means to customize them, it may still be necessary to change other parts of their behavior or interaction patterns due to the requirements stated in Architecture or Component Specification. These modifications are done by the *Component Assembler* as part of the Code within Implementation.

They may be accomplished by introducing a dedicated wrapper component that contains the existing component and exposes the required interfaces to the environment. The implementation of these visible interfaces uses functionality of the contained component in a way that fulfills the given specifications. This approach is only feasible if no fundamental modifications of the original component's behavior are necessary. ◁

▷ **Adaptation by Reimplementation:** Adapt the behavior of an existing component by reimplementing it according to the current specification.

Another, albeit drastic approach to adapt an existing component to a new context is to reimplement it from scratch. This may become necessary, for example, if the original component is not compatible with the intended technical platform, does not meet performance expectations or an initial wrapping approach seems too complex and error-prone. The required development efforts of the *Component Assembler* during the production of the Code are considerably higher compared to **Adaptation by Wrapping**. Still, the study of the original component's behavior and interfaces may provide valuable guidelines for the reimplementation. In this respect, **Adaptation by Reimplementation** is closely related to the inter-result pattern **Reverse Engineering**. ◁

Composition patterns, on the other hand, mention different possibilities for the composition of existing components. These patterns are similar to the the structural patterns of [GHJV95]. Other subresult patterns are involved with subresults like Search and Evaluation, which could be realized by performing a market survey, for example. Even further subresult patterns are test and integration patterns like **White-box Testing**, **Black-Box Testing**, **Review**, and so on. Due to the large number of conceivable subresult patterns, we currently do not attempt to propose a detailed and sufficiently complete pattern catalog, but only present one further, non-technical pattern for the subresult Component Assignment of the main result Specification:

▷ **Make-or-Buy:** Prepare and conduct the decision whether to develop a specified component as a part of the current project, as a separate in-house project, or to order it from an external component vendor.
⟶ *See appendix, Section B.4.* ◁

6 Related Approaches

There already exists established literature about process and organizational patterns [Cop97]. A vaguely related approach can be found, for example, in [MM97]. This book contains a rather inhomogeneous collection of patterns, most of them concerning various technical and architectural aspects in the context of CORBA systems. However, it does also contain the organizational pattern "**Software Development Roles**" which suggests a clear role structure with roles similar to those proposed in Section 3. The fundamental difference to our approach is that all our proposed patterns strongly *rely* on the presence of such a role structure. We are convinced that the separation of responsibilities by roles is mandatory to every real-world componentware project.

An important contribution in the area of process and organizatorial patterns has been made by Coplien who proposes a generative pattern language to describe existing and to 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 choosing the ideal size of an organization, the importance of finding the right communication structure, and the choice of an adequate role 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. Compared to most of Coplien's patterns, our patterns are more concrete because they are tailored to the context of componentware and rely on a pre-defined role and result structure. However, there are also overlapping areas, for example with the **Review** and **Prototype** patterns which are also present in our catalog. Other patterns, like **Code Ownership**, **Patron**, or **Organization Follows Location**, are orthogonal to our approach and may therefore be combined with the patterns proposed in this paper. Similar considerations apply to the approaches of Beedle [Bee98] and Cockburn [Coc97].

A closely related approach to a flexible software development process suited to componentware has very recently been proposed in the scope of the Catalysis methodology [DW98]. It allows for a choice of alternative "routes" through the development process which create and elaborate a given result structure, consisting of parts like a business model, a system specification or system design documents. These different work-products are semantically related to each other and may be checked for consistency and completeness. The development process itself is customized by applying process patterns on different levels of concern: Development context patterns like **Object Development from Scratch** or **Re-engineering** determine the overall sequence of the main development phases, similar to the project patterns proposed in this paper. Within each phase several local patterns like **Make a business model** or **Implement technical architecture** suggest possible development activities that may be applicable.

Based on a preview version of the pattern catalog, we believe the Catalysis patterns are mainly intended to provide general guidelines in applying this particular development method. In contrast, the patterns presented in Section 5 focus more on the development process itself and its implications in the context of componentware. The proposed result structure is more clearly defined and serves as a solid backbone when choosing and applying the individual patterns. The actual activity of producing a development result is not covered by our patterns (after all, the technical architecture *has* to be implemented at some point), allowing for more advanced application of development knowledge, featuring key aspects like reuse and definition of subprojects. Still, it is interesting that the common problem of defining a flexible development process model leads to convergent solutions relying on a defined result structure and appropriate process patterns. This seems to justify the chosen approach and encourages further work in this area.

7 Conclusion

We have presented a methodology for organizing the development process of component-oriented systems based on a given, standardized result structure together with a first, preliminary version of a pattern language. 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 presented patterns need to be expanded and improved. Especially example applications and counter-examples are urgently missing at this point, as well as application guidelines with detailed recommendations for tools, metrics, and techniques. We also plan to include economical aspects and implications in our approach, possibly by defining additional tasks and roles. The elaboration and consolidation of the pattern catalog is the focus of our current work.

References

[AG83] Albrecht and Gaffney. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE Transactions SE*, 9(6), 1983.

[BBHP95] Wolfgang Bartsch, Klaus Bergner, Rudolf Hettler, and Barbara Paech. **S**tudenten **E**ntwickeln **U**niverselles **H**ochschulinformationssystem: Erfahrungen aus einem Softwaretechnik-Praktikum. In *Proceedings of SEUH*. German Chapter of the ACM, Teubner-Verlag, 1995.

[Bee98] Michael Beedle. BPRPatternLanguage home page, `http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?BPRPatternLanguage`, 1998.

[Ber96] Klaus Bergner. Under pressure – recommendations for managing a practical course in software engineering. In *Proceedings of Software Engineering: Education and Practice '96*. IEEE Press, 1996.

[BH97] Klaus Bergner and Franz Huber. Systems development with Java: Experiences from a practical project course in software engineering. In *Proceedings of the Eighth International Workshop on Software Technology and Engineering Practice '97*. IEEE Press, 1997.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley & Sons, 1996.

[Boe81] B. W. Boehm. Constructive cost model. *Software Engineering Economics*, 1981.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2nd edition, 1994.

[Coc97] Alistair Cockburn. *Surviving Object-Oriented Projects*. Addison-Wesley, 1997.

[Cop94] J. O. Coplien. A development process generative pattern language. In *PLoP '94 Conference on Pattern Languages of Programming*, 1994.

[Cop97] Jim Coplien. OrgPatterns home page, `http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?FrontPage`, 1997.

[DW98] Desmond D'Souza and Allan Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. to appear, `http://www.iconcomp.com/catalysis`, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HS96] Brian Henderson-Sellers. The need for process, `http://www.sigs.com/publications/docs/oc/9612/oc9612.sellers.html`. *Object Currents – The monthly On-Line Magazine*, December 1996.

[Jac98] Ivar Jacobson. Component-based development using UML. Talk at SE:E&P'98, Dunedin, Newzealand, 1998.

[MM97] Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns*. Wiley Computer Publishing, 1997.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Software Components*. Springer-Verlag, 1997.

[Tec94] Technische Universität München, Institut für Informatik. *TIS Project Page* `http://tis.informatik.tu-muenchen.de/tis.html`, 1994.

[Tec95] Technische Universität München, Institut für Informatik. *FEEDBACK Project Page* `http://tis.informatik.tu-muenchen.de/stp95/`, 1995.

[Tec96] Technische Universität München, Institut für Informatik. *AutoFocus Project Page* `http://autofocus.informatik.tu-muenchen.de/`, 1996.

[Tec97] Technische Universität München, Institut für Informatik. *SimCenter Project Page* `http://autofocus.informatik.tu-muenchen.de/simcenter`, 1997.

[Tec98a] Technische Universität München, Institut für Informatik. *CARE Project Page* `http://autofocus.informatik.tu-muenchen.de/stp98/`, 1998.

[Tec98b] Technische Universität München, Institut für Informatik. *JAMES Project Page* `http://wwwbruegge.informatik.tu-muenchen.de/scp98/index.html`, 1998.

A Description Format

Intent A concise summary of the pattern's rationale and intent. It mentions the particular development issue or problem that is addressed by the pattern.

Also Known As Other possible names for the pattern, if any.

Motivation An illustration of the particular development issue or problem that is addressed by the pattern. If possible, a scenario is provided which motivates the use of the pattern.

Applicability A description of the context in which the pattern may be applied. It is largely based on the current state of the development project, but sometimes also external circumstanecs have to be considered.

Structure A visual representation of the pattern's structure based on the result structure as shown in Figure 1. The results involved in the pattern are emphasized with a grey background whereas grey arrows denote the involved relationships between them. The state of the development project is visualized by different fill styles of the involved subresults: a dark grey background indicates a completely developed result structure, a gradient fill represents a partially developed result structure whereas a white background denotes missing or irrelevant development information.

The suggested development activities within a pattern, namely, creating results and evaluating and establishing consistency criteria, are denoted by sequence numbers attached to the corresponding boxes and arrows.

Tasks and Roles A description of the participating tasks or roles as well as their responsibilities and interactions in application of the pattern. This section complements the structure diagram in illustrating the solution provided by the pattern.

Consequences A short discussion of the results, consequences and trade-offs associated with the pattern. It allows an evaluation of the pattern's usefulness.

Application Guidelines Practical guidelines, hints and techniques useful to apply the pattern. Particular methods, measures or tools are mentioned which support the application of the pattern.

Application Examples Known uses of the pattern in practical development projects. These application examples illustrate the acceptance and usefulness of the pattern, but also mentions counter-examples and failures.

Related Patterns A list of related patterns that are either alternatives or useful in conjunction with the described pattern.

B Selected Patterns

B.1 Architecture-Driven (Project Pattern)

Intent Organizing the overall development process by first establishing a system architecture. Starting out from this architecture, the developer tries to gather reasonable user requirements, designs appropriate business as well as technical components and builds an implementation strongly based on this architecture.

Also Known As Inside-Out

Motivation Certain application domains like telecommunication systems for example, are dominated by existing system architectures. Most of the available software components are meant to be used in the context of a given architecture. The overall development process should be adapted to this situation.

Applicability The application domain is dominated by a small number of system architectures. These architectures are well-understood, widely used, and stable. Ideally, they are open architectures and subject to international standardization (ANSI, ISO, OSI, etc.). The products of system development are not so much targeted at a single customer, but are likely to apply to a larger market.

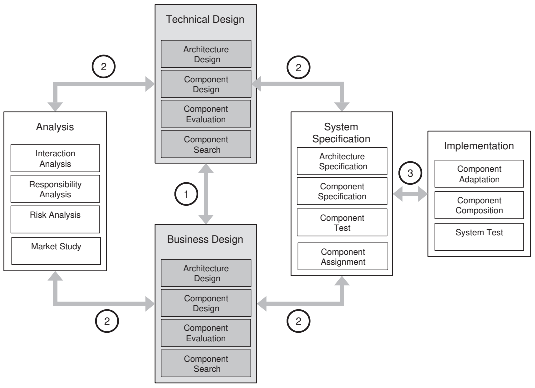

Figure 2: Structure of the Pattern **Architecture-Driven**

Tasks and Roles The *System Architect* has a clear understanding of the available technical and business-oriented architectures in the given application domain. He is responsible for searching and evaluating these architectures and the appropriate components. Although tentative customer requirements may exist due to market studies, for example, they are not the primary evaluation criterion. After a certain technical and business-oriented architecture has been chosen, the *System Architect* prepares the according design documents which serve as input to the results of Analysis, System Specification and finally Implementation.

While elaborating the Analysis documents, a product profile needs to be established according to the market demands. Once tentative requirements based on the given system architecture have been identified, it is possible to elicit concrete user requirements and supply them to the other development tasks.

Specification requires the integration of both technical and business-oriented architecture to produce a complete and consistent specification document which allows implementation of the system. The available components are mainly tested against predefined requirements of the chosen system architecture until the actual user requirements are provided by the results of Analysis. Note that there may even exist standardized test procedures provided by the original achitecture specification. Once the suitable components have been identified, they are either handed over to implementation within the current project or set up as clearly defined subprojects.

Consequences The quality of the products relies strongly on the chosen system architecture. Choosing a widely used or even standardized system architecture as the starting point of system development facilitates the production of capable and reliable software systems. The finished products or even part of the developed components apply to a wider market.

However, many of the existing standard architectures cover a large area and require a lot of effort to understand and use them adequately. It is probably necessary to design a whole line of products on a given architecture as a single product usually does not justify these efforts.

Furthermore, it is not advisable to cling to a given standard architecture if it is simply not appropriate for the development project in question. A careful evaluation procedure has to ensure the adequacy of the chosen architecture.

Application Guidelines If there exists a widely used or even standardized system architecture, it is likely that many supporting tools and frameworks have already been developed. Frameworks in particular are well suited for rapid application or prototype developement that helps to understand and evaluate the given architecture. However, they may not be appropriate for the actual system implementation as frameworks usually rely on a strong coupling between functionality and interaction. This is generally not a desired characteristic of component-oriented systems.

Application Examples Telecommunication systems, SAP R/3

Related Patterns Variants of this pattern are **Technical Architecture Driven** and **Business Architecture Driven** which focus on a dedicated part of the overall system architecture. The mentioned aspects and consequences of the architecture-driven approach, however, also apply to these variants.

The patterns **Experimental Business Prototyping**, **Experimental Technical Prototyping**, and **Combined Experimental Prototyping** may be applied to understand as well as to evaluate the conceivable architectures and available components. Based on these prototypes, it may be possible to analyze the development risks and obtain more concrete user requirements.

B.2 Experimental Prototyping (Main Result Pattern for Business Design or Technical Design)

Intent Building a preliminary throw-away version of the system or some of its vital components to get familiar with a possible architecture. Depending on the nature of the architecture, this pattern subsumes two subpatterns concerning business architecture and technical architecture, respectively. The experience gained has to be transferred to Architecture Design.

Also Known As —

Motivation Many tentative system architectures look promising in theory but are actually hard to implement and exploit in practice. Building an experimental prototype of the system or some of its vital components leads to a better understanding of the architecture and the way it should be used. This is especially important in the architecture-driven development approach which relies strongly on the appropriate system architecture. Furthermore, the time spent to develop the prototype may serve as an indicator of the total development time needed using the given architecture. For example, a project may involve tight requirements regarding the system's response times. An experimental technical prototype would be very useful to evaluate possible technical infrastructures with respect to these requirements.

Applicability One or several new system architectures have been proposed. These architecures are not widely used in the application domain or there exists little in-house experience and knowledge about them. Ideally, the specifications of the system or some of its vital components are known although it is possible to build a purely technical prototype without them.

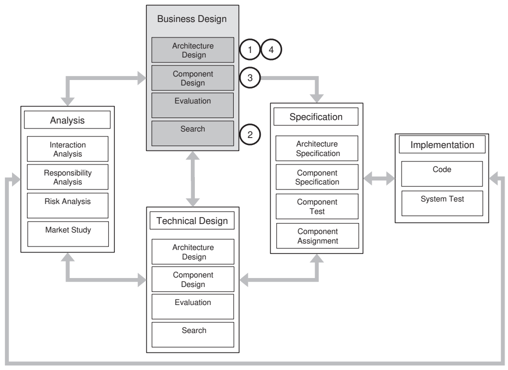

Figure 3: Structure of the Pattern **Experimental Technical Prototyping**

Tasks and Roles The *System Architect* makes a preselection of available, reasonable architectures and proposes a suitable one for the prototype in Architecture Design. Naturally, as the information regarding the architecture is rather incomplete at this early stage, the results of Search within Business Design and Technical Design comprise a set of components which only roughly meet the customer's requirements but allow the *Component Assembler* to build a prototype quickly. Experience gained from the prototype's development like, for instance, deficiencies or peculiarities of the architecture, time spent to implement and understand the architecture or degree and quality of tool support are collected in the results of Architecture Design, conceivably in the form of a Technical Experience Report. Note that the development documents of the experimental prototype itself should be kept in a separate result structure to prevent its reuse.

Consequences The practical experience gained by building an experimental prototype proves very valuable when having to decide for a certain system architecture. Many unexpected problems or deficiencies of a system architecture may be identified during the process of implementation. If the architecture is finally selected for product development, there already exists a solid understanding and knowledge about all relevant aspects. These experiences help to minimize the risk of designing the wrong system architecture. As an additional benefit, it may be possible to easily extend the experimental prototype into an explorative prototype which facilitates the analysis of customer requirements.

In general, the experimental prototype should not be used as the basis of further system development. It is built for purely technical or analytical reasons with the least possible effort and will, therefore, usually not meet the quality standards of a finished product.

Application Guidelines Any rapid prototyping tool based on the system architecture is helpful.

Application Examples —

Related Patterns Architecture-Driven, Explorative Prototyping

B.3 Explorative Prototyping (Main Result Pattern for Analysis)

Intent Build a preliminary throw-away version of the system or some of its vital components in order to explore and analyze the customer requirements.

Also Known As —

Motivation It is often very difficult to collect all relevant system requirements and state them unambigously in the Analysis documents. Presenting an early prototype of the system's GUI, for example, facilitates communication with the user about his requirements and may even lead to the discovery of new requirements that were not considered before.

The pattern **Explorative Prototyping** is generally useful in the context of a more requirements-driven approach, but it may also be be valuable in an approach focusing on existing components.

Applicability Although preliminary system requirements were collected, it is obvious that some of them are not refined enough or even inconsistent. There may be indications that some requirements have not been discovered yet.

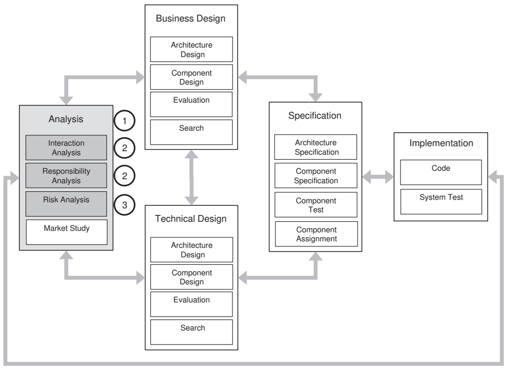

Figure 4: Structure of the Pattern **Explorative Prototyping**

Tasks and Roles The *System Analyst* has provided initial requirement specification documents within the subresults Interaction Analysis and Responsibility Analysis. They serve as the starting point for the implementation of a GUI or even functional prototype that represents a dedicated part of the main result Analysis. If there is good tool support, the development of the prototype may be done by the *System Analyst*, who is supported by the *Component Assembler*.

The explorative prototype is presented in discussions with the customer, and the resulting changes and additions to the required interactions and responsibilities are recorded in the respective subresults. It may be necessary to update the results of Risk Analysis due to additionally introduced requirements.

Note that the development documents of the explorative prototype itself should be kept in a separate result structure to prevent its reuse.

Consequences Using **Explorative Prototyping** leads to a better understanding and a more complete capturing of the system's requirements. This in turn minimizes the risk that the finished product does not meet the user's expectations.

In general, the explorative prototype should not be used as the basis of further system development. It is built for purely analytical reasons with the least possible effort and will therefore usually not meet the quality standards of the finished product.

Application Guidelines Any rapid prototyping tools suited to componentware are helpful.

Application Examples —

Related Patterns Requirements Driven, Experimental Prototyping

B.4 Make-or-Buy (Subresult Pattern for Component Assignment)

Intent Prepare and conduct the decision whether to develop a specified component as a part of the current project, as a separate in-house project, or to order it from an external component vendor.

Also Known As —

Motivation Once the results of Component Specification are sufficiently completed and stable, a decision about the further development of the specified components has to be reached during Component Assignment. In principle, there are three conceivable options for every concerned component:

1. Develop the component as part of the current project.
2. Develop the component as a separate in-house project.
3. Order the component from an external component vendor.

As an example, consider the development of a specialized tree-view GUI component that is needed to present hierarchically structured application data. It seems reasonable to realize such a specific GUI component entirely within the current project as part of a normal system development. However, it may be justified to set up a new project for this component because this particular kind of tree-view is likely to be reused in other projects as well. This approach allows a dedicated team of developers within the same company to build the component in parallel to normal system development. It is also conceivable that such a tree-view is already available on the market or there are well-known vendors of GUI components who are willing to accept a development order. In this case it could be advisable to draw the tree-view component from them.

This decision is not straightforward as it depends on technical, economical and strategical considerations that must be carefully balanced against each other.

Applicability The Specification documents for the component in question have to be complete, consistent and stable. Moreover, both Evaluation and Search subresults of Business Design and Technical Design should be sufficiently elaborated to judge the market situation. Moreover, there may also exist preliminary Component Test results about existing components. Finally, the specified component itself is demanding and important enough to justify the efforts of a make-or-buy decision.

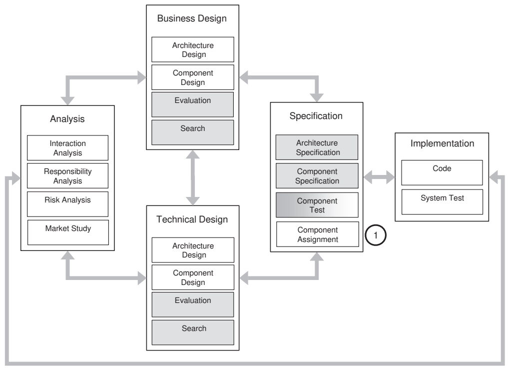

Figure 5: Structure of the Pattern **Make-or-Buy**

Tasks and Roles When performing a make-or-buy decision, the *System Architect* and the *Project Coordinator* have to consider three main areas of concern:

Technical Aspects If the component's functionality is sufficiently generic and reusable or important as part of a larger technical infrastructure, it is advisable to set up a new project by transferring the results of Specification to the Analysis documents of the new result structure. This allows to decouple further development of the component and to facilitate its reuse with the extensive amount of dedicated development information available.

Economical Aspects The definition as a separate in-house project allows to speed up development if the required resources for this project are available. This speed-up may be very important if the time-to-market for the developed system is critical. Even further speed-up and possibly substantial cost reductions may be achieved if the component in question may be purchased or ordered from an external vendor. The component vendor is likely to possess specific know-how and resources to quickly develop (or deliver) a component of high quality with respect to efficiency, reliability, etc. Still, cost estimations which compare expected in-house versus external development costs should be performed by applying an appropriate method.

Strategical Aspects Depending on the overall strategy of the developing company, it may be advisable to keep the development of a particular component in-house. If the required infrastructure or development know-how fall within the core competences of the company, it is not desirable to source out this subproject. Moreover, if the component in question is critical for the developed system, it seems reasonable to realize it in-house in order to minimize the risk of failure and dependance on external partners.

Consequences In-house development of a given component leads to important potential benefits: The knowledge and experience gained during development may result in a competetive edge on the market, and reduces the dependence on external partners. Producing a component in-house may even allow a company to act as a component vendor itself and to earn additional money by selling the component in question.

However, the decision to buy a component from a dedicated vendor implies different potential benefits: The necessary investment in know-how and resources may be saved for other parts of the development project. As the component is likely to be sold in higher numbers by the component vendor, it is probably cheaper compared to in-house development. In case of standard components which are also offered by different vendors, the dependence on a single vendor is diminished as well.

Application Guidelines The application of adequate and effective cost estimation methods are obviously vital for performing a make-or-buy decision. Although there already exist a number of established methods like Function Point Analysis [AG83] or Constructive Cost Model (COCOMO) [Boe81], it is likely that methods which are specifically tailored to componentware will be available in the future.

The strategical aspects of a make-or-buy decision with respect to market ratings and determination of core competences should be supported by managerial-economics: it is likely that the results and experiences gained in other areas like the car industry may partly be transferred to software engineering.

Application Examples —

Related Patterns —