Process Patterns for Component-Based Software Development

Ehsan Kouroshfar, Hamed Yaghoubi Shahir, and Raman Ramsin

Department of Computer Engineering
Sharif University of Technology
kouroshfar@ce.sharif.edu, yaghoubi@ieee.org, ramsin@sharif.edu

Abstract. Component-Based Development (CBD) has been broadly used in software development, as it enhances reusability and flexibility, and reduces the costs and risks involved in systems development. It has therefore spawned many widely-used approaches, such as Commercial Off-The-Shelf (COTS) and software product lines. On the other hand, in order to gain a competitive edge, organizations need to define custom processes tailored to fit their specific development requirements. This has led to the emergence of process patterns and Method Engineering approaches.

We propose a set of process patterns commonly encountered in component-based development methodologies. Seven prominent component-based methodologies have been selected and reviewed, and a set of high-level process patterns recurring in these methodologies have been identified. A generic process framework for component-based development has been proposed based on these process patterns. The process patterns and the generic framework can be used for developing or tailoring a process for producing component-based systems.

Keywords: Component-Based Development, Software Development Methodologies, Situational Method Engineering, Process Patterns.

1 Introduction

Although *Component-Based Development* (CBD) is not a novel approach, it is still extensively used for building various types of software systems, and is expected to remain popular for the foreseeable future. There exist several software development methodologies that support the construction of component-based systems, and the domain has matured over the years. When viewed collectively, CBD methodologies have indeed addressed all the relevant issues; however, none of the methodologies covers all the aspects of component-based software development. A general methodology can resolve this through addressing the deficiencies while being customizable according to the specifics of the project situation at hand. An alternative approach to tackling this problem is *Assembly-Based Situational Method Engineering* (SME), in which a bespoke methodology is constructed according to the characteristics of the project situation at hand. The construction process involves selecting and assembling reusable process *fragments* from a repository [1, 2].

G.A. Lewis, I. Poernomo, and C. Hofmeister (Eds.): CBSE 2009, LNCS 5582, pp. 54–68, 2009.

Process patterns were first defined as "the patterns of activity within an organization (and hence within its project)" [3]. Later definitions focused on defining patterns as practical process chunks recurring in relevant development methodologies. One such definition, focusing on the object-oriented paradigm, defines a process pattern as "a collection of general techniques, actions, and/or tasks (activities) for developing object-oriented software" [4]. Process patterns are typically defined at three levels of granularity: *Tasks*, *Stages* and *Phases* [4]. A *Task* process pattern depicts the detailed steps to perform a specific task. A *Stage* process pattern includes a number of *Task* process patterns and depicts the steps of a single project stage. These steps are often performed iteratively. Finally, a *Phase* process pattern represents the interactions between its constituent *Stage* process patterns in a single project phase. *Phase* process patterns are typically performed in a serial manner.

Since process patterns describe a process fragment commonly encountered in software development methodologies, they are suitable for being used as *process components*. They can thus be applied as reusable building blocks in an assembly-based SME context, providing a repository of process fragments for assembling processes that are tailored to fit specific projects or organizational needs. The OPEN Process Framework (OPF) is an example of using process patterns for general method engineering purposes [5]. Sets of process patterns can also be defined for specific development approaches; the *object-oriented* process patterns of [4], and the *agile* process patterns proposed in [6] are examples of domain-specific patterns, and have indeed inspired this research.

Although a number of process patterns have been introduced in the context of component-based development [7], a comprehensive set of patterns providing full coverage of all aspects of component-based development has not been previously proposed. We propose a set of process patterns commonly encountered in component-based development. The patterns have been identified through studying seven prominent component-based methodologies. A generic process framework, the *Component-Based Software Development Process* (CBSDP), has been constructed based on these process patterns. The generic framework and its constituent process patterns can be used for developing or tailoring a methodology for producing component-based systems. It should be noted that the proposed framework is not a new methodology for component-based development, but rather a generic pattern-based SME model for CBD: method engineers can instantiate the generic framework and populate it with instances of the constituent process patterns according to the particulars of their CBD projects. This approach is already prevalent in methodology engineering frameworks such as OPEN/OPF [8], SPEM-2 [9], and the Eclipse Process Framework (EPF) [10]; indeed, the proposed framework and process patterns can be defined and used as method plug-ins in the *Eclipse Process Framework Composer* (EPFC) tool.

The rest of the paper is structured as follows: Section 2 provides brief descriptions of the seven CBD methodologies used as pattern sources; Section 3 contains the proposed generic framework for component-based software development (CBSDP); in Section 4, the proposed phase-, stage-, and task process patterns are explained; Section 5 validates the patterns through demonstrating how the proposed patterns correspond to the methodologies studied; Section 6 contains the conclusions and suggestions for furthering this research.

2 Pattern Sources: Component-Based Software Development Methodologies

The seven methodologies that have been studied for extracting process patterns are: *UML Components*, *Select Perspective*, *FORM*, *KobrA*, *Catalysis*, *ASD*, and *RUP*. These methodologies were selected because they are well-established and mature, and also because adequate resources and documentation are available on their processes. We briefly introduce each of these methodologies in this section.

UML Components is a UML-based methodology aiming to help developers use technologies such as COM+ and JavaBeans for defining and specifying components [11]. The process shows how UML can be used in a CBD context; that is, to define components, their relations, and their use in the target software system.

Select Perspective was the result of combining the object modeling language introduced in [12] with the Objectory use-case-driven process (later integrated into RUP). In its original form, it incorporated business modeling, use case modeling and class modeling activities. A CBD extension was later added, addressing business-oriented component modeling, component modeling of legacy systems, and deployment modeling. *Select* is based on a service-oriented architecture adapted from the Microsoft Solution Framework application model. It constitutes of three types of services: user services, business service, and data services.

Feature-Oriented Domain Analysis (FODA) was introduced in 1990, and presented the idea of using features in requirement engineering [13]. It was later extensively used in several product-line engineering methods. One such method is the *Feature-Oriented Reuse Method* (FORM) [14, 15], which has added the features of architectural design and construction of object-oriented components to *FOD*A, thus providing a CBD methodology.

The *KobrA* methodology is based on a number of advanced software engineering technologies, including product-line engineering, frameworks, architecture-centric development, quality modeling, and process modeling [16, 17]. These methods have been integrated into *KobrA* with the specific aim of systematic development of high quality, component-based software systems.

The *Catalysis* methodology is a component-based approach based on object-oriented analysis and design [18]. It has been influenced by other object-oriented methodologies such as Syntropy and Fusion. *Catalysis* provides a framework for constructing component-based software.

The *Adaptive Software Development* (ASD) methodology was introduced in 1997 [18], and is the only Agile method that specifically targets component-based development.

The *Rational Unified Process* (RUP) is the most well known of the selected methodologies [18]. As a third-generation object-oriented methodology, RUP is use-case-driven and architecture-centric, and incorporates specific guidelines for component-based development. It should be noted, however, that RUP has evolved into a method engineering framework (*Rational Method Composer* – RMC); a further testimony to the changing nature of software engineering, stressing the importance of process patterns in this new trend.

3 Proposed Component-Based Software Development Process (CBSDP)

A thorough examination was conducted on the selected methodologies, as a result of which, 4 *phase* process patterns, 13 *stage* process patterns, and 59 *task* process patterns were identified.

The process patterns have been organized into a generic process framework for CBD methodologies. This framework, which we have chosen to call the Component-Based Software Development Process (CBSDP), is shown in Figure 1. CBSDP consists of four phases (each considered a phase process pattern): *Analysis*, *Design*, *Provision*, and *Release*.

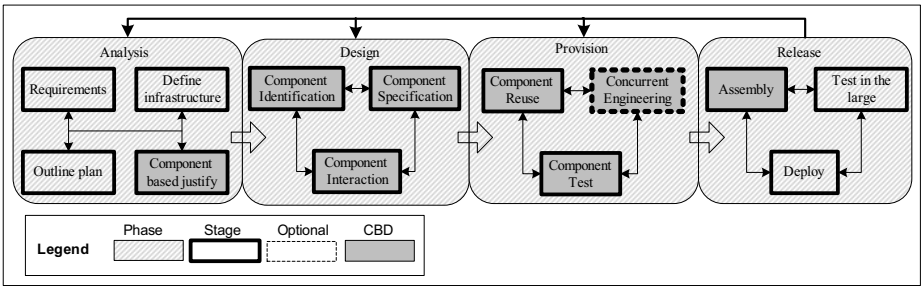

Fig. 1. The proposed Component-Based Software Development Process (CBSDP)

The process begins with the *Analysis* phase, in which the requirements of the system are elicited first. The applicability of the component-based approach to the project at hand is then investigated; after all, the component-based approach may not be suitable for the project. The infrastructure of the project is then defined, and a preliminary project plan and schedule is outlined. In the *Design* phase, the components of the system are identified; based on the interactions among these components, complete specifications are defined for each component. In the *Provision* phase, components are classified into two categories: Those which are retrieved from a repository of reusable components, and those which are constructed from scratch. Components are then developed and tested individually. In the *Release* phase, components are assembled together to form the system proper. After system testing is conducted, the end-product will be deployed into the user environment.

It is worth noting that CBSDP is a framework that guides the method engineer when assembling instances of process patterns. Not all stage process patterns in CBSDP are mandatory; pattern selection is based on the needs of the project situation at hand. For example, suppose that a method engineer needs to construct a methodology for a product line project that is similar to a previous project. Hence, the *Component-based Justify* stage will not be required in the *Analysis* phase. In addition, the *Concurrent Engineering* stage in the *Provision* phase could be omitted, since it may be possible to use existing components instead of constructing new ones.

4 Proposed *Stage* and *Task* Process Patterns

In this section, details of the *Stage* process patterns constituting each of the four aforementioned phases are discussed. Furthermore, some of the constituent *Task* process patterns are briefly reviewed in the descriptions. As previously mentioned, these process patterns can be used independently by method engineers to extend/enhance an existing process, or to construct a new component-based software development methodology.

It is important to note that the arrows connecting the task process patterns do not imply any sequencing. Although it is logical to assume that some of the tasks will precede others, we do not intend to impose any such ordering. The main purpose of this approach is to enable the method engineer to decide the order of the tasks based on the specifics of the project situation at hand.

4.1 Requirements

Requirements Engineering is where the high level *Requirements* of the target system are defined (Figure 2). The inputs to this stage include the *Project Vision* and the *Customer's Viewpoints* on the different aspects of the project. The key factor in component-based development projects – as emphasized in many component-based methodologies such as Select Perspective, UML Components, and KobrA – is how to use previous experience and existing components in future projects. In order to support this purpose, *Existing Components* and *Previous Projects' Assets* are two important inputs to this stage.

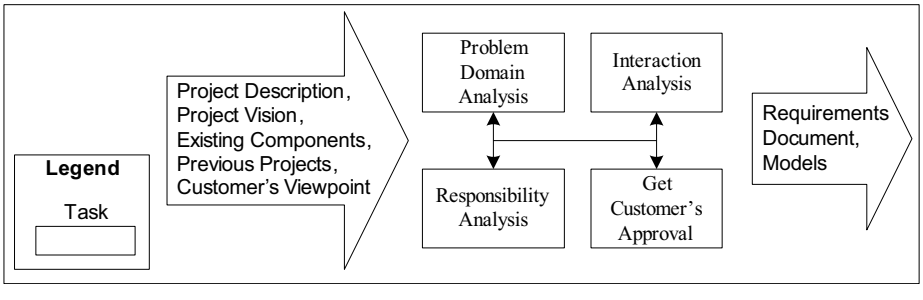

Fig. 2. Constituents of the Requirements stage process pattern

Requirements are defined during the *Problem Domain Analysis* task. During *Interaction Analysis*, the interactions of the system with its environment are examined. The system boundary and the external actors interacting with the system are thus determined. In the *Responsibility Analysis* task, the main functionalities of the system are investigated, and functional and non-functional requirements are identified through active collaboration with the customer. The customer's approval is obtained through the *Get Customer's Approval* task.

4.2 Define Infrastructure

The *Project Infrastructure* should be defined at the beginning of the project. As shown in Figure 3, it includes *Standards*, *Teams Definition*, *Tools* used during the project, and the *Development Platform*. The tailored version of the development process to be used in the project is also determined, with *Method Constraints* (covering modeling, testing, build, and release activities) duly defined. In product-line engineering and component-based projects, the *Project Infrastructure* is a key product because it can be used in various projects. Similarly, the infrastructure of previous projects can be tailored for use in future projects.

The inputs to this stage are the requirements extracted in the *Requirements* stage, along with previous experiences compiled in the *Previous Projects* document, *Existing Infrastructure,* and the *Project Plan*. As a result, the *Project Infrastructure Document* and *Team Definition* are produced. The requirements document may be refined during this stage.

In cases where the organization imposes predefined standards and development platforms, the *Select Standards* and *Specify Development Platforms* tasks are not applicable. These tasks have therefore been defined as optional.

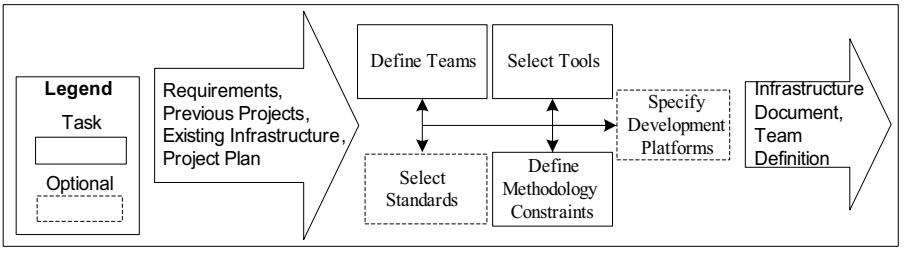

Fig. 3. Constituents of the Define-Infrastructure stage process pattern

4.3 Outline Plan

In this stage of the CBSDP (Figure 4), the initial project management documents (such as project scope, time schedule, etc.) are produced. The management documents are used and updated during the development lifecycle.

The inputs to this stage are the *Requirements Document*, *Infrastructure Document*, *Existing Components*, and *Project Objectives*. Preliminary estimation of time and resources and the definition of the project scope are also performed during this stage. An initial viable schedule is then outlined for the project. Based on the team definition provided in the infrastructure document, the work units are assigned to team members. Initial risk assessment activities are also conducted in this stage.

In component-based projects, there is a need to study the market in order to obtain information about existing systems, similar solutions, and reusable components. This information can be used in time and resource estimation, and also when preparing an initial schedule for the project.

The *Project Plan*, *Management Document* and *Risk Assessment* are the deliverables of this stage. The management document contains all the information needed for managing the project. It includes the project plan and schedule (among other things), and may be refined during later stages of the project.

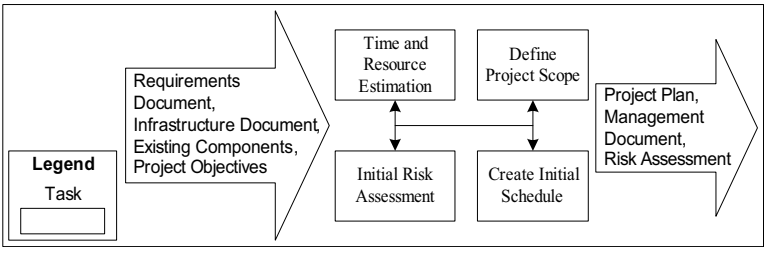

Fig. 4. Constituents of the Outline-Plan stage process pattern

4.4 Component-Based Justify

This stage determines whether the component-based approach can be used on the project at hand (Figure 5). It is a critical stage of the analysis phase since it checks if the project makes sense in a component-based context. It makes use of the *Requirements Document*, *Risk Assessment Document*, *Existing Components* and *Previous Projects* experience to produce the *Feasibility Study* and the *Business Case*, which justifies the project financially.

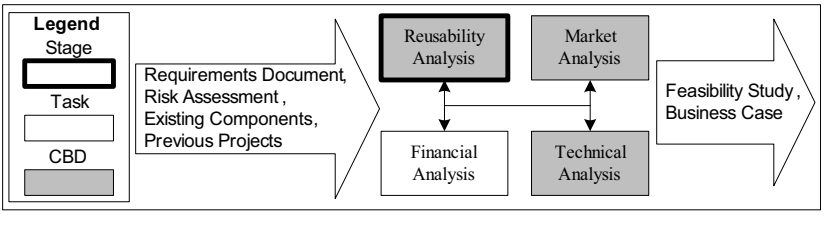

Fig. 5. Constituents of the Component-based-Justify stage process pattern

During the *Technical Analysis* task, we investigate whether it is possible to build the system and realize all of its functional features on the basis of components. Since *Reusability Analysis* contains various tasks, it is defined as a stage process pattern. In this stage, the component repository is first checked in order to assess the applicability of existing components; furthermore, the system components which are to be produced during the current project are explored as to their potential reusability. During the *Market Analysis* task, we search for similar components and systems available on the market. We also investigate the marketability of the components developed for the new system. Based on this information, *Financial Analysis* is performed to assess the economic feasibility of the project.

4.5 Component Identification

The second phase, *Design*, starts with the *Component Identification* stage (Figure 6). It accepts the *Requirements Document*, *Existing Interfaces*, *Existing Components*, *Previous Projects*, and the *Business Case* as input. The main goal of this stage is to create an initial set of interfaces and component specifications. It determines the *System Interfaces* from the interactions between the system and its environment. *Business Interfaces* and *Technical Interfaces* are also identified in this stage. *Business Interfaces* are abstractions of all the information that should be managed by the system, while *Technical Interfaces* manage the technical components (such as database components). The *Business Type Model* contains the specific business information that must be maintained by the system, and will be used in the *Component Specification* stage. Furthermore, an *Initial Architecture* and *Components Specification* are defined in this stage. At the end of this stage, the *Business Type Model*, *System Interfaces*, *Business Interfaces*, and *Component Specs and Architecture* will be produced.

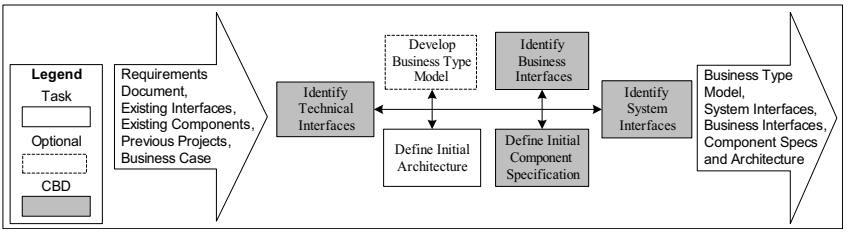

Fig. 6. Constituents of the Component-Identification stage process pattern

4.6 Component Interaction

After defining an initial set of components and their interfaces, we should decide how these components should work together. This task is performed in the *Component Interaction* stage (Figure 7), which uses and refines the *Business Interfaces*, *System Interfaces*, *Technical Interfaces*, and *Component Specifications and Architecture*.

Business operations needed by system to fulfill all of its expected functionality are defined. Existing design patterns are used for refining the *Interfaces* and *Component Specs*. Certain optimization criteria, such as minimization of calls, removal of cyclic dependencies, and normalization of operations and interfaces should be considered during this refinement.

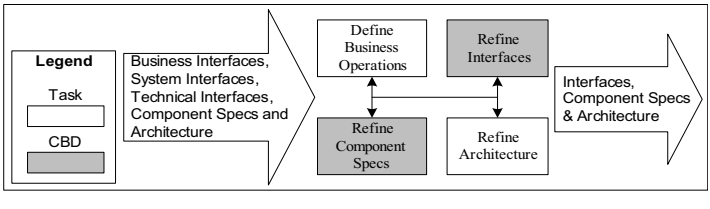

Fig. 7. Constituents of the Component-Interaction stage process pattern

4.7 Component Specification

This is the last stage of the *Design* phase; this means that all the information needed for building the components should be provided to the development teams. In this stage (Figure 8), *Design by Contract* is applied through specifying preconditions and postconditions for class operations. A precondition is a condition under which the operation guarantees that the postcondition will be satisfied. Furthermore, constraints are added to components and interfaces to define how elements in one interface relate to elements in others. Furthermore, *Business Rules and Invariants* are added to the component specifications. Finally, interfaces will be merged or broken in order to provide simpler interfaces.

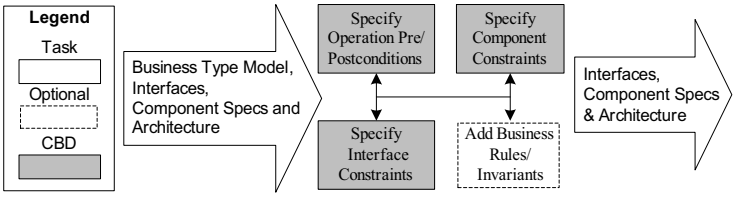

Fig. 8. Constituents of the Component-Specification stage process pattern

4.8 Component Reuse

The *Component Reuse* stage (Figure 9) determines a specific strategy for providing the components which can be constructed or purchased. All components should be classified according to this strategy. For example, one strategy is to build *Business Components*, and purchase all the others. Based on the acquisition strategy, component specifications can be issued to the organization's development teams, commissioned to trusted partners, or sought from commercial sources.

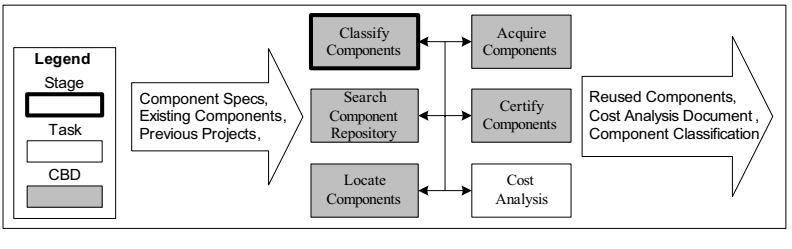

Fig. 9. Constituents of the Component-Reuse stage process pattern

When a candidate component arrives, it undergoes formal certification, which results in acceptance or rejection. If the component is certified, it will be stored in the component repository, and then published for access by developers in different projects. Component developers first search the component repository for components they require. Whenever components are discovered, they will be retrieved and examined, perhaps even tested, before they are reused. *Cost Analysis* is performed to determine which

alternative is economically preferable: to build a new component or to reuse an existing one. At the end of this stage it would be clear which component should be constructed from scratch and which one should be reused.

4.9 Concurrent Engineering

As depicted in Figure 1, the *Concurrent Engineering* stage is an optional stage, since we may be able to reuse existing components instead of constructing new ones. Components to be built are assigned to development teams and are concurrently constructed in this stage (Figure 10). Since teams work in a parallel fashion, the development pace is sped up considerably. Each team first *Defines Cycles* for implementing the components assigned to it. Implementation is performed iteratively according to component specifications in fixed time-boxes. Test code and documentation can be written in tandem with coding the components. Code inspection, with the aim of code refactoring and optimization, may also be done during the *Components Implementation* task.

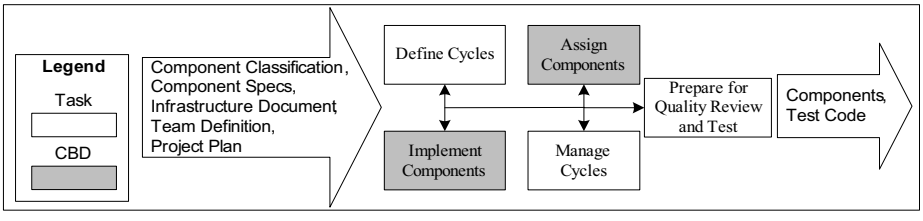

Fig. 10. Constituents of the Concurrent-Engineering stage process pattern

It is important to note that each team manages the project pace through continuous monitoring and control during the *Concurrent Engineering* stage. Keeping the cycles on track is the main concern of the *Manage Cycles* task. At the end of the stage, components should be prepared for quality review and testing. The *Concurrent Engineering* stage uses *Component Classification*, *Component Specs*, *Infrastructure Document*, *Team Definition* and the *Project Plan* to produce the new components.

4.10 Component Test

The *Component Test* stage (Figure 11) focuses on the verification and validation of the components. Documents, code and models should be tested and verified. The goal of *Component Testing* is to ensure that the components are ready to be delivered. *Component Test* is not a system-level test, and mainly consists of black box testing, unit testing and regression testing.

The *Requirements Document* and *Component Specs* are the inputs to this stage. After defining the *Test Plan*, *Test Cases* are either selected from the test base or generated according to *Component Specs*. The next step is to run the test cases on different components and record the results. Code inspection and review can also be conducted with the purpose of code refactoring, but this is not mandatory in all projects. During *Fix Bugs* tasks, minor defects would be resolved, but major errors should be addressed in the Concurrent Engineering stage.

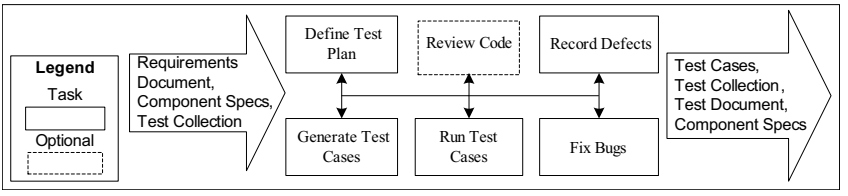

Fig. 11. Constituents of the Component-Test stage process pattern

4.11 Components Assembly

Assembly is the process of putting components and existing software assets together in order to build a working system. User interfaces of the system, which satisfy the user requirements, should then be designed (Figure 12). This stage shares many characteristics with standard configuration practices. Each individual component can be viewed as a separate configuration item, and the *Components Architecture* represents the system configuration definition.

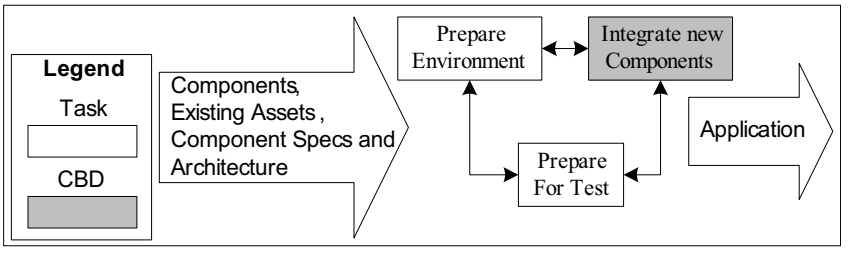

Fig. 12. Constituents of the Components-Assembly stage process pattern

The environment should first be prepared for installing the new component. Then the current system, which is integrated with the new component, will be prepared for testing. Components can be assembled simultaneously or incrementally based on the project strategy.

4.12 Test in the Large

System-level testing is conducted in this stage. The goal of *Test in the Large* (Figure 13) is to prove that the application is ready to be deployed. Defects found in this stage are either resolved by the *Fix Bugs* task, or referred to the *Provision* phase. The tasks of this stage are very similar to the *Component Test* stage, with one difference: the planning and generation of test cases is based on system-level testing strategies. *User Test* is an important stage during which the whole system is validated by users. Important tasks such as *Acceptance Testing* and *Quality Assurance* are conducted during this stage. The system will be ready to be deployed in the working environment after the completion of *Test in the Large* stage.

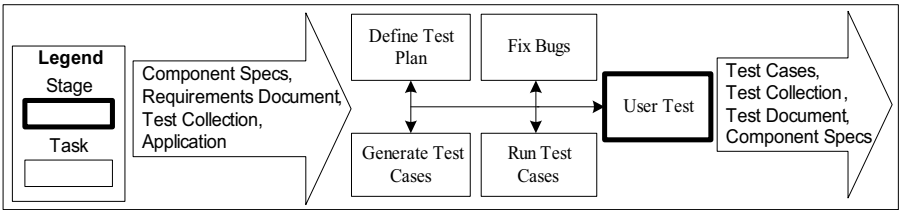

Fig. 13. Constituents of the Test-in-the-Large stage process pattern

4.13 Deploy

The aim of this stage (Figure 14) is to deploy the developed system into the user environment. The environment should first be set up for the deployment of the new system. Providing the necessary software and hardware is a prerequisite to the installation of the developed system. System users should then be prepared for working with the new system. To this aim, user manuals and appropriate application documents are provided, and users at different organizational levels are trained. The new system is then deployed. Deployment tasks should be performed with due attention to the project *Infrastructure Document*. In addition, components which have been tested and validated so far are added to the components repository for future reuse.

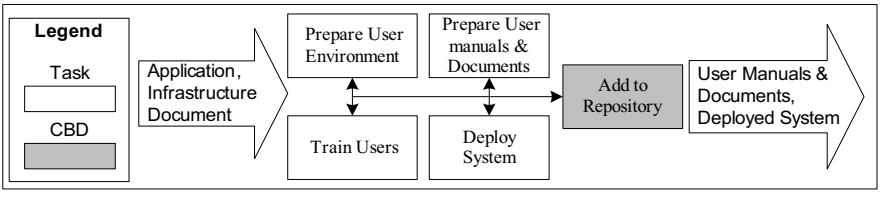

Fig. 14. Constituents of the Deploy stage process pattern

5 Realization of Proposed Process Patterns in Component-Based Methodologies

In this section, we demonstrate how different phases of the component-based methodologies studied can be realized by the proposed process patterns. Table 1 shows how the phases of the seven methodologies used as pattern sources correspond to the proposed stage process patterns. The results seem to indicate that the process patterns are indeed valid as to the coverage of activities that are typically performed in component-based development. In other words, all of these methodologies can be engineered by using the proposed framework and process patterns. It can therefore be concluded that the stated framework and process patterns are valid, although it is possible to enrich the repository by adding new process patterns.

Table 1. Realization of proposed process patterns in the source CBD methodologies

Methodologies	Phases	Corresponding Stage Process Patterns
UML Components	Requirements	Requirements, Define Infrastructure, Outline Plan, Component-based Justify
	Specification	Component Identification, Component Interaction, Component Specification
	Provisioning	Component Reuse, Concurrent Engineering, Component Test
	Assembly	Assembly
	Test	Test in the Large
	Deployment	Deploy
Select Perspective	Supply	Component Specification, Component Reuse, Concurrent Engineering, Component Test, Assembly, Test in the Large
	Manage	Component Reuse, Component Test
	Consume	Requirements, Outline Plan, Component Identification, Component Interaction, Deploy
FORM	Feature Modeling	Requirements
	Architecture Design	Component Interaction
	Architecture Refinement	Component Interaction
	Candidate Object Identification	Component Identification
	Design Object Modeling	Component Identification, Component Interaction
	Component Design	Component Specification
KobrA	Context Realization	Requirements
	Specification	Requirements, Component Identification
	Realization	Component Identification, Component Interaction, Component Specification
	Implementation & Building	Concurrent Engineering
	Component Reuse	Component Reuse
	Quality Assurance	Component Test, Assembly, Test in the Large
	Incremental Development	Concurrent Engineering, Assembly
ASD	Project Initiation	Requirements, Define Infrastructure, Outline Plan
	Adaptive Cycle Planning	Outline Plan, Component Identification, Component Interaction, Component Specification
	Concurrent Component Engineering	Component Reuse, Concurrent Engineering, Component Test, Assembly
	Quality Review	Test in the Large
	Final Q/A and Release	Test in the Large, Deploy
Catalysis	Requirements	Requirements, Define Infrastructure, Outline Plan, Component-based Justify
	System Specification	Requirements, Component Identification
	Architectural Design	Component Identification, Component Interaction
	Component Internal Design	Component Specification, Component Reuse, Concurrent Engineering, Component Test, Assembly, Test in the Large
RUP	Inception	Requirements, Define Infrastructure, Outline Plan
	Elaboration	Requirements, Component Identification
	Construction	Component Interaction, Component Specification, Component Reuse, Concurrent Engineering, Component Test
	Transition	Assembly, Test in the Large, Deploy

6 Conclusions

We have proposed a generic process framework for component-based software development. Within this general process, we have proposed a number of process patterns that are commonly used in component-based development. Extraction of these process patterns was based on a detailed review of seven prominent component-based development methodologies.

After the identification of the process patterns, they were checked against the source methodologies to demonstrate that they do indeed realize the phases of the

methodologies. The results, as depicted in Table 1, seem to verify that the patterns do indeed cover the activities performed in the source methodologies.

The proposed process patterns can be used in component-based development projects for engineering a custom process tailored to fit the requirements of the project at hand.

The process patterns proposed in this paper were detailed at the phase and stage levels. Further work can be focused on detailing the task process patterns introduced. Through completing the specifications of the task process patterns, it will be possible to set up a repository of component-based development process patterns to enable assembly-based situational method engineering. The general process presented in this paper can be used as a template for defining component-based processes. A method engineer can use this general template and populate it with specialized instances of the proposed process patterns, thus instantiating a new component-based software development methodology. To this end, work is now in progress on defining our proposed framework and process patterns as method plug-ins in the *Eclipse Process Framework Composer* tool.

Acknowledgement. We wish to thank the ITRC Research Center for sponsoring this research.

References

1. Mirbel, I., Ralyté, J.: Situational Method Engineering: Combining Assembly-based and Roadmap-driven Approaches. Requirements Engineering 11(1), 58–78 (2006)
2. Ralyté, J., Brinkkamper, S., Henderson-Sellers, B. (eds.): Situational Method Engineering: Fundamentals and Experiences. In: Proceedings of the IFIP WG 8.1 Working Conference, Geneva, Switzerland, September 12-14. IFIP International Federation for Information Processing, vol. 244. Springer, Boston (2007)
3. Coplien, J.O.: A Generative Development Process Pattern Language. In: Pattern Languages of Program Design, pp. 187–196. ACM Press/ Addison-Wesley, New York (1995)
4. Ambler, S.W.: Process Patterns: Building Large-Scale Systems Using Object Technology. Cambridge University Press, Cambridge (1998)
5. Henderson-Sellers, B.: Method Engineering for OO Systems Development. Communications of the ACM 46(10), 73–78 (2003)
6. Tasharofi, S., Ramsin, R.: Process Patterns for Agile Methodologies. In: Ralyté, J., Brinkkemper, S., Henderson-Sellers, B. (eds.) Situational Method Engineering: Fundamentals and Experiences, pp. 222–237. Springer, Heidelberg (2007)
7. Bergner, K., Rausch, A., Sihling, M., Vilbig, A.: A Componentware Development Methodology based on Process Patterns. In: 5th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois (1998)
8. Firesmith, D.G., Henderson-Sellers, B.: The OPEN Process Framework: An Introduction. Addison-Wesley, Reading (2001)
9. Object Management Group: Software and Systems Process Engineering Metamodel Specification v2.0 (SPEM), OMG (2007)
10. Haumer, P.: Eclipse Process Framework Composer, Eclipse Foundation (2007)
11. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, Reading (2003)

12. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs (1991)
13. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
14. Kang, K.C., Lee, J., Donohoe, P.: Feature-Oriented Product Line Engineering. IEEE Software 9(4), 58–65 (2002)
15. Sochos, P., Philippow, I., Riebisch, M.: Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In: Weske, M., Liggesmeyer, P. (eds.) NODe 2004. LNCS, vol. 3263, pp. 138–152. Springer, Heidelberg (2004)
16. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-Based Product-Line Engineering with UML. Addison-Wesley, Reading (2001)
17. Atkinson, C., Bayer, J., Laitenberger, O., Zettel, J.: Component-based Software Engineering: The KobrA Approach. In: 22nd International Conference on Software Engineering (ICSE 2000), 3rd International Workshop on Component-based Software Engineering, Limerick, Ireland (2000)
18. Ramsin, R., Paige, R.F.: Process-Centered Review of Object-Oriented Software Development Methodologies. ACM Computing Surveys 40(1), 1–89 (2008)