Component-based Development Process and Component Lifecycle

Ivica Crnkovic[1], Michel Chaudron[2], Stig Larsson[3]
[1]*Mälardalen University, Department of Computer Science and Electronics, Sweden*
[2]*Eindhoven University of Technology, Dept. of Mathematics and Computing Science, The Netherlands*
[3]*ABB, Corporate Research, Sweden*
[1]*ivica.crnkovic@mdh.se,* [2] *m.r.v.chaudron@TUE.nl,* [3]*stig.larsson@mdh.se*

Abstract

The process of component- and component-based system development differs in many significant ways from the "classical" development process of software systems. The main difference is in the separation of the development process of components from the development process of systems. This fact has a significant impact on the development process. Since the component-based approach is a relatively young approach in software engineering, the main emphasis in the area has been in development of technologies, while process modeling is still an unexplored area. This paper analyses the basic characteristics of the component-based approach and its impact on the development process and lifecycle models. The generic lifecycle of component-based systems and the lifecycle of components are discussed, and the different types of development processes are discussed in detail: architecture-driven component development, product-line development and COTS-based development. Finally a short case study illustrates the principles and specifics of component-based processes.

1. Introduction

There exist many models for software development processes and lifecycles. Most of them are specified considering some specific, often non-technical goals, such as quality, predictability, dependability, or flexibility, and are often independent of technology. Examples of such models are different sequential models such as Waterfall or V model, or iterative modules such as spiral model, or different agile methods, or standard and de-facto standards such as ISO 9000, or CMMI. These models are usually specified in general terms and they require tailoring for particular projects. Some development processes and life-cycle models have their origins in a technology or in a particular approach. A very characteristic example is Object-Oriented Development (OOD) which emprises both technologies and processes. RUP (Rational Unified Process) has a clear influence of OOD.

Component-based software engineering (CBSE), as a young discipline is still focused on technology issues: modeling, system specifications and design, and implementation. There is no established component-based development process. Yet many principles of component-based development (CBD) have significant influence on the development- and maintenance process and require considerable modifications of standard development processes.

This paper discusses specifics of the component-based approach and its impact on component-based development processes and we illustrate this by discussing adaptations of a specific process model. In continuation we identify and discuss three different types of component-based development processes: (i) Architecture-driven component development, (ii) Product-line development and (iii) COTS-based development. Finally we present a case study from industry which clearly shows a paradigm shift from a process with an emphasis on programming to emphasis on requirements and component management, and tests and verification activities. The rest of the paper is organized as follows. Section two provides a general framework of lifecycle processes. Section three presents the basic principles of CBD and their impact on lifecycle processes. Section four discusses three different approaches and processes in CBD. Section five discusses some of the approaches through a case study. Finally the last section concludes the paper and gives directions for further research.

2. Lifecycle Process Models for Software Systems

Every product, including software products, has a lifecycle [1]. Although lifecycles of different products may be very different, they can be described by a set of phases or stages that are common for all lifecycles. The phases represent the major product lifecycle periods and they are related to the state of the product.

Figure 1 shows a frequently encountered example of products lifecycle phases [1]: concept, development, production, utilization and retirement.

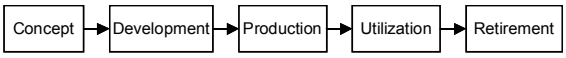

Figure 1: Generic Product Lifecycle

Software products have a slightly different lifecycle: typically the production phase can be neglected as a separate phase as the production activities are considerably smaller than other activities. Also, since software is easy to change (although the consequences of a change may be severe and may require a lot of effort) it is often developed and released in different versions. This allows concurrent operation and development. The model from Rajlich and Bennett [2] takes into consideration these characteristics, and defines the software lifecycle slightly different from the product lifecycle model: The concept phase, including the initial design and development, is called initial development. The production phase is omitted since it is assumed to be a part of a development phase. The utilization phase including further development is actually a series of evolution- and servicing cycles. Finally the retirement phase is divided into a phase-out and close-down phase.

During the initial development phase the first functioning version of the product is developed from scratch to satisfy initial requirements. During the evolution phase the quality and functionality of the product is iteratively extended. At certain intervals new versions of the product are released and delivered to the customers. In the servicing phase only minor defects in the product are repaired. The phase-out phase the product is still used but not serviced any more. Finally during the close-down phase the product is withdrawn from the market: either replaced by another product or disposed.

Very often development organizations perform the same activities in the initial development phase as in each evolution cycle. In this way, an existing software product will evolve into a next version by repeating the same sequence of phases - although probably with different emphasis. These activities grouped together define a software development lifecycle [3]. Not all software development lifecycle models are suitable for all types of software systems. Usually large systems which include many stakeholders and whose development lasts a long period prefer using sequential models. Systems which use new technologies, which are smaller, and to which the time-to market is important, usually explore evolutionary models. These models are more flexible and can show results much earlier than sequential models.

How well these models suit the development of component-based systems is an open question. Can they be applied directly or is some adoption required to match the principles of component-based approach? Let us discuss that in the following sections.

3. Component-based approach

The main idea of the component-based approach is building systems from already existing components. This assumption has several consequences for the system lifecycle;

- Separation of development processes. The development processes of component-based systems are separated from development processes of the components; the components should already have been developed and possibly have been used in other products when the system development process starts.
- A new process: Component Assessment. A new, possibly separated process, finding and evaluating the components appears. Component assessment (finding and evaluation) can be a part of the main process, but many advantages are gained if the process is performed separately. The result of the process is a repository of components that includes components' specifications, descriptions, documented tests, and the executable components themselves.
- Changes in the activities in the development processes. The activities in the component-based development processes are different from the activities in non-component-based approach. For the system-level process, the emphasis will be on finding the proper components and verifying them. For the component-level process, design for reuse will be the main concern.

To illustrate the specifics of the component-based development processes we shall use the Waterfall model - the simplest one – as a reference. However, the illustration can relatively simply also be applied to other development processes. Figure 2 shows the main activities of the Waterfall model: Requirements Specification, Analysis & Design, Implementation, Test, Release and Maintenance.

The primary idea of the component-based approach is to (re)use the existing components instead of implementing them whenever possible. For this reason already in the requirements and design phases the availability of existing components must be considered.

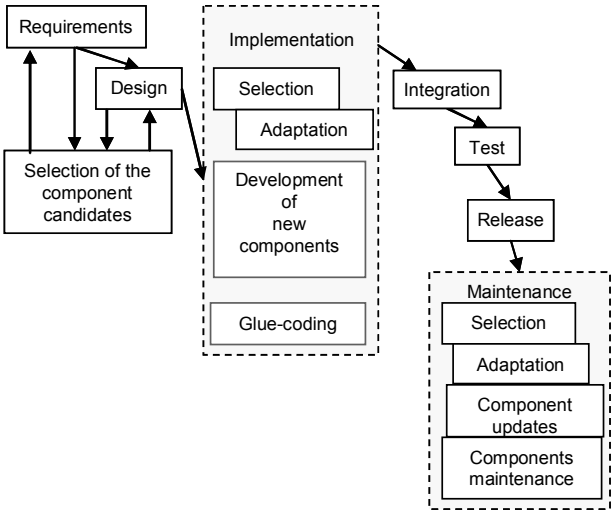

Figure 2: Component-based Waterfall Software product lifecycle

The implementation phase will include less coding (in an ideal case no coding) for implementing functions, and focus more on selection of available components, and if necessary their adaptation to the requirements and design specification. The required functionality that is not provided by any existing component must be implemented, and in a component-based approach the relevant stakeholders (for example the project manager, the organization management, system architects) will consider whether these new functions will be implemented in the form of new components that can be reused later. An inevitable part of the implementation of a component-based system is the glue-code which connects the components, enables their intercommunication and if necessary solves possible mismatching. In the ideal case, glue code can be generated automatically; otherwise it has to be developed in addition to the components that are selected.

Figure 2 still shows a simplified and an idealized process. Its assumption is that the components selected and used are sufficiently close to the units identified in the design process, so that the selection and adaptation process require (significantly) less effort than the components implementation. Further, the figure shows only the process related to the system development – not to the supporting processes: Assessment of components and the component-development process. Actually, there might be many parallel component development processes. These processes are depicted in Figure 3.

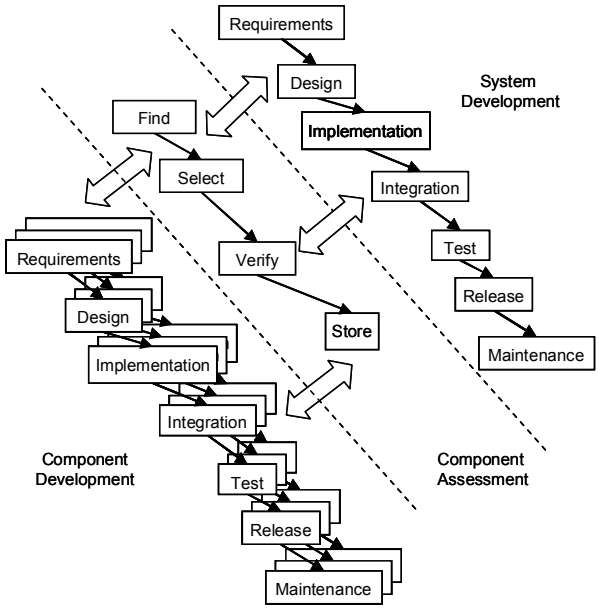

Figure 3: Parallel processes of component-based development

The processes shown in Figure 3 can be performed independently of each other, but certainly there are activities that bridges these processes: Which components will be a subject for searching, what type of verification is required, which verified components do exist – these are similar questions as staring points of the component assessments which is initiated from the system development process. Similarly, the questions such as which functions will be provided by the components being developed, which requirements will be posed on the components, for which type of systems these components will be used, are related to the component requirements. How these "crosscutting" activities will be implemented, and how these processes will be integrated, depends on type of component-based process. This will be discussed in the section 4.

First, we shall discuss the activities of each process.

3.1 Component-based system development process

We shall shortly discuss the activities shown in figures 2 and 3.

Requirements Phase

In a non-component-based approach a requirements specification is an input for development of the system. In a component-based approach this is somewhat different; the requirements specification will also consider the availability of existing components; the requirements should correlate to the assortment of the

components, i.e. the requirements specification is not only input to the further development, but also a result of the design and implementation decisions.

Analysis & Design Phase

The design phase of component-based systems follows the same pattern as a design phase of software in general; it starts with a system analysis and a conceptual design providing the system overall architecture and continues with the detailed design. From the system architecture, the architectural components will be identified. These components are not necessarily the same as the implementation components but they should be identified and specified in a detailed design as assemblies of the existing components. Again, as in the requirements process, a tradeoff between desired design and a possible design using the existing components must be analyzed. In addition to this, there will be many assumptions that must be taken into consideration: For example, it must be decided which component model(s) will be used, which will have impact on the architectural framework as well as on certain system quality properties.

Implementation Phase

As shown in Figure 2, the implementation activities only partially consist of coding – actually the more pure a component-based approach is achieved, the less coding will be present. The main emphasis is put on component selection and its integration into the system. This process can require additional efforts. First the selection process should ensure that appropriate components have been selected with respect to their functional and extra-functional properties. This requires verification of the component specification, or testing of some of the component's properties that are important but not documented. Second, it is a well known fact [4] that even if isolated components function correct, an assembly of them may fail, due to invisible dependencies and relationships between them, such as shared data shared resources. This requires that components integrated in assemblies are tested before they are integrated into the system.

The adaptation of components may be required in order to avoid architectural mismatches (such as incompatible interfaces), or to ensure particular properties of the components or the system. There are several known adaptation techniques, such as parameterized interface, wrappers and adapters.

Integration Phase

In a non-component-based development process the integration phase includes activities that build the systems from the incoming parts. The integration phase does not include "creative" activities in the sense of creating new functions by production of new code, and for this reason it is desired to automate and rationalize the process as much as possible. The phase is however very important as it is the "moment of truth"; many problems become visible due to architectural mismatches of the incoming components, or due to unwanted behaviour of different extra-functional properties on the system level. That is why the integration phase is tightly connected to the system test phase in which the system functions and extra-functional properties are verified.

In a component-based approach many integration parameters are determined by the choice of component technology, and component selection. The component technology standardizes the architectural frameworks, reuses architectural patterns, and usually provides means for efficient integration. For this reason the integration process should be more straightforward and less error-prone. This holds when considering architectural mismatch of the components, but the verification of extra-functional properties (in particular emerging properties, i.e. properties that are not visible on component level, but exist on the system level), remains complex and is in many cases as difficult as for non-component-based systems.

Since system functions are not exclusively realized by components alone but often by a set of components, verification of these functions requires that the components are integrated before the entire system is built. For this reason the integration phase for component-based systems development is spreading to earlier phases: implementation, design and even into the requirements phase.

Test Phase

In CBD a need for component verification is apparent since the system developers typically have no control over component quality, component functions, etc., as the component could have been developed in another project with other purposes. The tests performed in isolated components are usually not enough since their behaviour can be different in the assemblies and in other environments [5]. The component test is actually performed at many different times: though individual assessment, when integrated in an assembly or subsystem, and when deployed (integrated) into the systems (see Figure 4).

Release Phase

The release phase includes packaging of the software in forms suitable for delivery and installation. The CBD release phase is not significantly different from a "classical" integration.

Maintenance Phase

In everyday life one of the patterns of products maintenance is: Repair the product by replacing the malfunctioning component. The objective of the component-based approach for software is similar: A system should be maintained by replacement of components. The characteristics of physical (hardware) components is however different from software components. While hardware components can be exposed to a process of degradation in functionality and quality, software components do not degrade in this manner. In principle there should be no need for their change. However, experience shows the opposite: Lehman's well known law [5] says: "The entropy of a system increases with time unless specific work is executed to maintain or reduce it." – i.e. the software system will degrade if not maintained. The reason is not the degradation of the software itself but because of the changes of the environment the software operates in. Even if a system functions properly, as time goes by, it has to be maintained. The approach of CBD is to provide maintenance by replacing old components by new components or by adding new components into the systems. The paradigm of the maintenance process is similar to this for the development: Find a proper component, test it, adopt it if necessary, and integrate it into the system (see Figure 4).

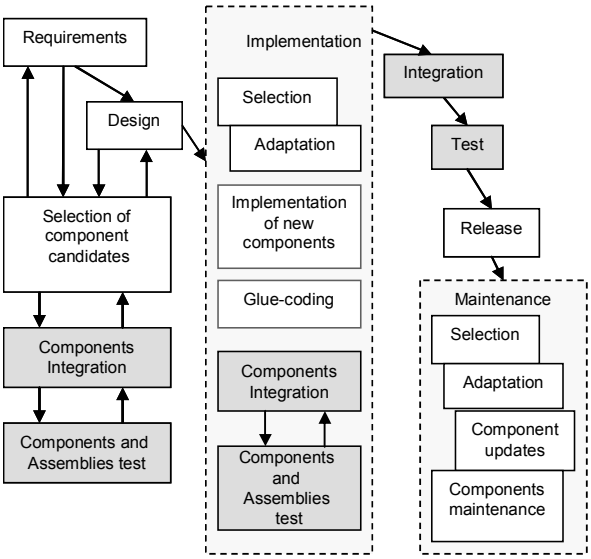

Figure 4: Integration and test in several phases of the CBD process

3.2 Component assessment

While development of component-based systems significantly decreases the detailed design and implementation efforts during the system development, it requires additional efforts in other activities. For example instead of implementing required functions, developers have to search for components that provide such functionality. Further they must verify that the selected components i) provide (or approximate) the desired functionality, and ii) that the components can successfully be integrated with other components. The consequence can be that not the best components (i.e. components that provide the "best functions") can be selected, but the components that fit together.

To make the system development process efficient (i.e. to achieve better time-to-market) many assessment activities can be performed independently and separately from the system development.

A generic assessment process includes the following activities:

Find – Find, from an unlimited component space, the components that might provide the required functionality. This functionality can be a part of the system being developed, or of a system (or systems) plan to be developed.

Select – The candidate components found, are compared and ranked. A component that is most suitable for the given requirements and constraints is selected. The ranking of components should be maintained throughout system development such that alternatives for a function can quickly be found.

Verify – Verification is part of component selection. The first level of verification includes testing functional and certain extra-functional properties of a component in isolation. A second level of verification includes testing the component in combination with other components integrated in an assembly.

Store – when a component is assumed to be a good candidate for the current and/or future applications, it is stored in a component repository. The repository should not only include the component itself, but also additional information (metadata) that can be useful in further exploitation of the component. Examples of such data are: measured results of component performance, known problems, response time, tests passed and tests results.

These activities in the component assessment process are not necessary performed in the order as shown. Depending on different architectural approaches (see section 4) some activities will be more important and will require more efforts, while some other will be very small or non-existing. For example, if a company uses only internally developed components, the "find" activity will not be significant as the components are stored in internal repositories.

3.3 Component development process

The component development process is in many respects similar to system development; requirements must be captured, analyzed and defined, the component must be designed, implemented, verified, validated and delivered. When building a new component, developers will reuse other components and will use similar procedures of component evaluation as for system development. There is however a significant difference: components are intended for reuse in (many) different products, many of them yet to be designed. The consequences of these facts are the following:

There is greater difficulty in managing requirements; Greater efforts are needed to develop reusable units; Greater efforts are needed for providing component specifications and additional material that help developers/consumers of the components.

We highlight here the specific characteristics of activities of a component development and maintenance process.

Requirements Phase

Requirements specification and analysis is a combination of a top-down and bottom-up process. The requirements elicitation should be the result of the requirements specification on the system level. However, since the components are built also for future, not yet existing, or even not planned systems, the system requirements are not necessary identified. For this reason the process of capturing and identifying requirements is more complex, it should address ranges of requirements and the possible reusability. Reusability is related to generality, thus the generally of the components should be addressed explicitly.

Analysis & Design Phase

The input to the design phase in the component development process comes from system design, system constraints and system concerns. Since such systems do not necessary exist, or are even not yet planned, the component designer must make many assumptions about the system. Many assumptions and constraints will be determined by selecting a component technology. This determines, for example, possible component interactions, certain solutions built in the technology (like transactions or security mechanisms), and assumptions of the system resources (like scheduling policies). For this reason, it is most likely that at design time (or earlier) a component model and a component technology that implements that model must be chosen.

For a component to be reusable, it must be designed in a more general way than a component tailored for a unique situation. Components intended to be reused require adaptability. This will increase the size and complexity of the components. At the same time they must be concrete and simple enough to serve particular requirements in an efficient way. This requires more design and development effort. According to some experiences, developing a reusable component requires three to four times more resources than developing a component which serves a specific purpose [6].

Implementation Phase

Implementation of components is determined very much by the component technology selected. Component technology provides support in programming languages, automation of component compositions, can include many services and provide many solutions that are important for the application domain. Good examples of such support are transactions management, database management, security, or interoperability support for distributed systems provided by component technologies such as .NET, J2EE, or COM+. Object-oriented languages are suitable for implementation of components since they provide mechanisms that efficiently support concepts of CBD. Examples of these elements are the Interface-concept in Java or virtual classes in C++.

Integration Phase

Components are built to be easily integrated into systems. For this reason integration considerations must be continuously in focus. Further integration with other components in an assembly, in order to provide a particular service, or generate a unit of test, is also possible. Actually the integration activities may be performed frequently – for example for test purposes. Usually component technology provides good support for components integration, and integration is being performed on daily basis.

Test Phase

Test activates are of particular importance because of two reasons. (i) The component should be very carefully tested since its usage and environment context is not obvious. No specific conditions should be taken for granted, but extensive tests and different techniques of verification should be performed. (ii) It is highly desirable that the tests and test results are documented and delivered together with the component to system developers.

Release Phase

Release and delivery of the components are the phase where (assemblies of) components are packaged into packages that are suitable for distribution and installation. Such a package will not only include the executable components, but also additional information and assets (meta-date that specifies different properties,

additional documentation, test procedures and test results, etc.).

Maintenance Phase

A specific issue of maintenance in component-based systems is the relation components-system. If a bug in a component is fixed, the question is, to which systems a new version of the components should be delivered. Who will be responsible for the update: the system of the component producer? Further, there is also a questions who will be responsible for component maintenance; is this a responsibility of the component producer, or the system producer? Is it supposed that the component producers have the obligation to fix the bugs and support its update in the (possibly) numerous systems, or that they can provide support with additional payment, or they do not provide any support at all. Even more difficult problems can be related to so-called "blame analysis". The problem is related to a manifestation of a fault and the origin of the fault itself. An error can be detected in one component, but the reason can be placed in another. For example due to a high frequency of input in component A, the component A required more CPU time, so that component B does not complete its execution during the interval assumed by Component C which provides a time-out error, and a user of the component C gets the impression that an answer from Component C was not delivered. The first analysis shows that he problem is in the component C, then B, then A, and finally in the input to A. The questions is who performs this analysis if the producers of components A, B and C are not the same. Such situations can be regulated by contracts between the producers and consumers of the components, but this requires additional efforts, and in many cases it is not possible at all.

These examples show that maintenance activities can be much more extensive that expected. For this reason it is important that the component producers build up a strategy on how to perform maintenance and take corresponding action to ensure the realization of this strategy. For example, the component producers might decide to provide maintenance support and then it is important that they reproduce the context in with the error was manifested.

4 Different architectural approaches in component-based development

The industrial practice has established several approaches in using component-based development. These approaches, while possibly similar in using component technology, can have quite different processes, and different solutions on the architectural level. Let us look to three approaches, all component-based, but with quite different assumptions, goals and consequently processes.

- Architecture-driven component development
- Product-line development
- COTS-based development.

Architect-driven component development, described in more details in [7], uses a top-down approach; components are identified as architectural elements and as a means to achieve a good design. Components are not primary developed for reuse, but to fit into the specified architectures. Component-based technologies are used, but this is mainly because of the extensive support of component technology for modelling and specification, for easier implementations, for the possibility of using standard service of a component technology. The main characteristic of these components is composability, while reusability and time-to-market issues are of less concern. The parallel development processes shown on Figure 3 are reduced to two semi-parallel processes – system development and component development (see Figure 5).

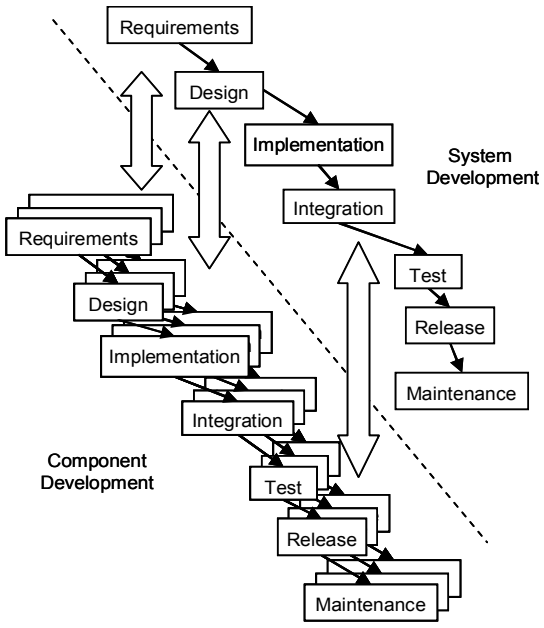

Figure 5. Architecture-driven component development process

Product-line development aims to enable efficient development of many variants of product, or family of products. Product-line development has a strategy to achieve a large commercial diversity (i.e. producing many variants and many models of products) with a minimal technical diversity at minimal costs. These

approaches are heavily architecture-driven, as the architectural solution should provide the most important characteristics of the systems. Within a given architecture (so called reference or platform architecture) component-based approach plays a crucial role – it enables reuse of components, and efficient integration process. So here composability, reusability and time-to-market are equally important. What is characteristic for product line is that the architectural solutions have direct impact on the component model. The component model must comply with the pre-defined reference architecture. Indeed in practice we can see that many companies have developed their own component model that suits their proprietary architecture. A second characteristic of product-line architecture (as a result of the time-to-market requirement) is a high degree of parallelism of the component development process and product development process and a combination of a top-down and bottom-up procedures. Referring to Figure 4 we can see that all three processes (system development, component assessment and component development) exist, but somewhat changed.

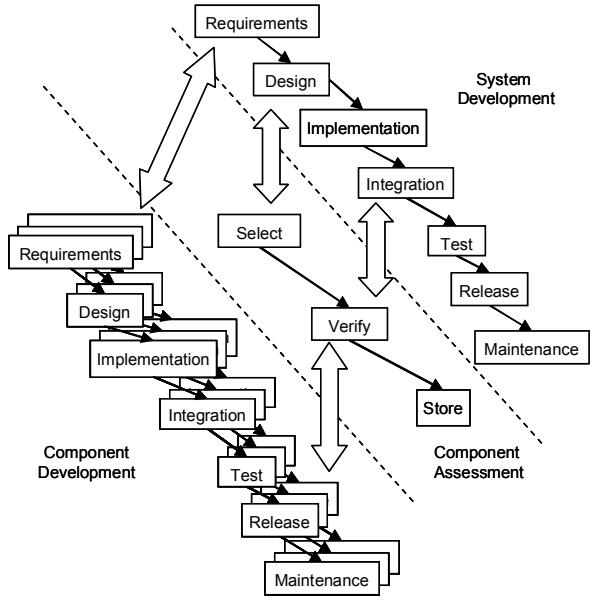

Figure 6. Product-line development

COTS-based development assumes that component development processes are completely separated from the system development process. The strongest concern is time-to-market from the component consumer point of view, and reusability from the component developer point of view. While the COTS approach gives an instant value of new functionality, (a lack of) composability may cause a problem if the COTS components do not comply to a component model, if the semantics are not clear or if architectural properties of the components are not properly and adequately documented. For COTS-based development, component assessment plays a much more important role than in the previous two approaches. Of al three approaches discussed here, Figure 4 most closely presents the COTS-based approach.

Which of these approaches are best, or most CBD-specific? There is no definitive answer. While COTS-based development looks like the most inherited to CBD approach, and by this the most promising, the practice in last decade has not shown big successes; on the other hand, after a surge of enthusiasm in both industry and research, the COTS components market has decreased and does not show revolutionary improvement. One of the reasons for that is that it is difficult to achieve reusability by being very general, effective, simple and at the same time provide attractive functionality. Furthermore, there are problems of trustworthiness (who can guarantee that the component is correct?), component verification and certification. The product line approach has been successful in many domains and its combination with CBD is a promising approach. Possible threats are increasing costs for development and maintenance of the component technologies developed internally. This includes compilers, debuggers, and in integrated development environments. In some cases the internally developed component technologies are replaced by the widely-used general-purpose component technologies, while keeping the overall product-line approach.

5. Case study in product-line development

To illustrate a product-line architecture process we discuss a process model used in a large international company in consumer electronics. The development divisions of the company are placed in four different countries and they produce numerous products with different variants and models. The company has adopted a component-based development approach using a product-line architecture. The component model and its supporting development tools are developed internally. The main reason for this are the specific requirements of the application domain: low resource usage, high availability, and soft real-time requirements.

The component model follows the basic principles of CBSE: The components are specified by interfaces which distinguish “require” from “provide” interfaces. In addition to functional specification, the interface includes additional information; the interaction protocols, the timeliness properties, and the memory usage. The component model enables a smooth

evolution of the components as it allows existence of multiple interfaces. The model has a specific characteristic; it allows hierarchical compositions: a composite component can be treated as a single component and can again be further integrated in other components. Components are also developed internally, but their development is separated from the development of the products.

The product-line architecture identifies the basic architectural framework. The product architecture is a layered architecture which includes (i) operating system, (ii) the component framework which is an intermediate level between domain-specific services and operating, (iii) core components which are included in all product variants, and (iv) application components that are different for different product variants. Complementary to this horizontal layering there is a vertical structuring in the form of subsystems. Subsystems are also related to the organizational structures; they are responsible for development and maintenance of particular components.

In the overall process there area three sets of the independent parallel processes: (i) An overall architecture and platform development process responsible for delivering new platforms and basic components, (ii) Subsystem development processes which deliver a set of components that provide different services, and (iii) the product development process which is basically an integration process. This process arrangement makes it possible to deliver new products every six months, while the development of subsystem components takes typically between 12 and 18 months. The specifics of these projects are that all deliverables have the same form. A deliverable is a software package defined as a component. The overall process that includes parallel development projects which deliverables are components and products is shown on Figure 7.

The development processes in our case is mainly of an evolutionary model. The platform, the subsystems and the products are developed in several iterations until the desired functionality and quality is achieved. This requires synchronizations of iterations.

Although the overall development and production is successful, the company faces several challenges. The most serious problem is late discovery of errors: The causes of errors are interface- or architectural mismatches or insufficient specifications of semantics of the components. Also the problems related to encapsulation of a service in components often occur; due to functional overlaps, or some requirements that affect the architecture, it is difficult to decide in which components a particular function will be implemented.

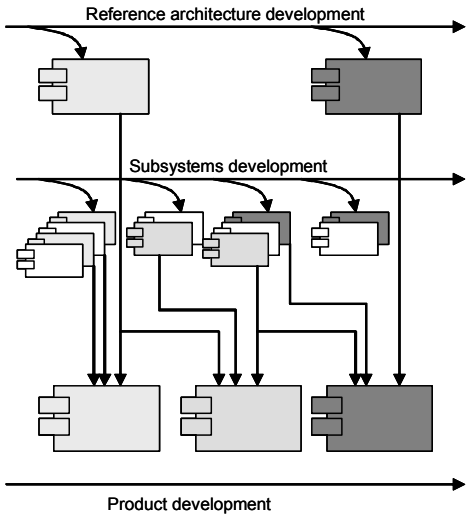

Figure 7. Products and components development processes

All these problems point out that it is difficult to perform the processes completely independently; negotiation between different subsystems and agreements about many technical details between different teams are necessary. For these reasons continuous coordination is necessary between development projects developing components and products. This reflects to the project and company organization. The following stakeholders have special roles in the projects:

- The system architect and management have overall responsibilities for requirements, policies, product line architecture, products visions, and long term goals.
- The project architect has a responsibility for the overall project which results in a line of products. He/she coordinates the architectural design of the product family and subsystems.
- The test and quality-assurance (QA) managers have similar role in their domains: to ensure coordination and compatibility of tests and quality processes.
- The subsystem architects are responsible for the designs of their subsystems and coordinate the design decisions with the project architect and architects of other subsystems.
- Each subsystem has a test team and a QA manager whose responsibility is the quality of the delivered subsystem components.
- An integration team is responsible for the delivery of a project and is represented by a product architect and QA- and test managers who coordinate the activities with other projects.

We can observe that the project teams have many "non-productive" stakeholders. This is in line with the component-based approach – more efforts must be put on overall architecture and test, and less on the implementation itself.

5. Conclusion and future work

In this paper we have described different phases of component-based system life cycle. These phases are described in a framework of a particular process model, but similar principles are valid for any other development processes. The main characteristic of component-base development processes is a separation (and parallelization) of system development from component development. This separation has a consequence on other activities: Programming issues (low-level design, coding) are less emphasized, while verification processes and infrastructural management requires significantly more efforts. We have seen that a component-based approach does not only require different expertise but also organizational changes in an enterprise.

This paper is a starting point for further research which is needed to establish principles of component-based processes. The work will continue in two parallel, but strongly related issues; (i) The goal of the first initiative is to specify an ideal (and yet realistic) life cycle process model for component-based systems.

The starting point of the research are component-based approaches itself, their technological characteristics and the possibilities they provide; (ii) The goal of the second initiative is to adopt and integration of different development processes and life-cycle models and methods (such as agile methods, iterative and incremental processes) with component-based procedures. The work will combine analytical and experimental approaches including extensive case studies.

6. References

[1] ISO/IEC 15288, System Engineering - System Life Cycle Processes, First Edition, ISO/IEC, 2002

[2] Rajlich, Bennett. *A Staged Model for the Software Life Cycle*. IEEE Computer, July 2000

[3] Kruchten, Philippe. *A Rational Development Process*, Crosstalk, July 1996

[4] Kurt Wallnau, "Dispelling the Myth of Component Evaluation" in Ivica Crnkovic and Magnus Larsson (editors), "Building Reliable Component-Based Software Systems", Artech House Publisher, 2002

[5] M M Lehman, Feedback in the Software Evolution Process**,** Keynote Address, CSR Eleventh Annual Workshop on Software Evolution: Models and Metrics. Dublin, 7 - 9th Sept. 1994, Workshop Proc., Information and Software Technology, sp. is. on Software Maintenance, v. 38, n. 11, 1996, Elsevier, 1996, pp. 681 – 686

[6] Szyperski C., *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[7] I. Crnkovic and M. Larsson, CBSE – Building Reliable Component-based Systems, Artech House, 2003