7

Component-based Development Process and Component Lifecycle

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 chapter begins from this premise and discusses its implications. The generic lifecycle of component-based systems and the lifecycle of components are presented first. The following different types of development processes are then discussed in detail: architecture-driven component development, product-line development and COTS-based development. These three types require different approaches and the use of different techniques and methods in all phases of the development process. The chapter describes these approaches, recommending the most suitable - and appropriate techniques -in each case and the implications of their use.

7

Component-based Development Process and Component Lifecycle

7.1 Introduction

For a successful development a technology only is not enough! Any slightly more complex project requires management of different aspects that are beyond a technology. Examples of such aspects are project planning, coordination between project stakeholders, management of resources, organization of work, and similar. In a product life-cycle (i.e. all phases in a product's life) technologies are enablers to particular technical solutions, but also catalysts for different development processes. These processes may be result of particular business or market requirements. Indeed this is true in the case of component-based development. Business and market requirements are drivers of component-based approach. Component-based technologies enable distributed development, parallel development, separation of the development process, increase reusability, etc., which are solutions to the demands on short time-to-market, lower costs or increased flexibility.

There exist many models for software (and systems) development processes and life-cycles. 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 adjustments 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, 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 CBD have significant influence on the development and maintenance process and require considerable modifications of standard development processes.

This chapter discusses specifics of component-based approach and its impact on component-based development processes and we illustrate this by discussing adaptations of a specific process model (waterfall model). In continuation we identify three different types of component-based development processes: Architecture-driven component development, Product-line development and COTS-based development. Finally we presents a case study from industry which clearly shows a paradigm shift from programming-dominated processes to requirements and component management, and tests and verification activities.

7.2 Lifecycle Process Models for Software Systems

Every product, including software products, has a lifecycle [ISO02]. 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 [ISO02]: concept, development, production, utilization and retirement.

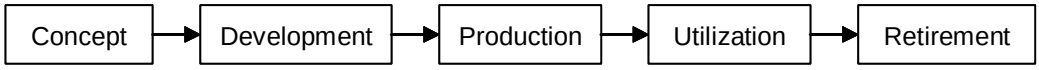

Figure 1: Generic Product Lifecycle

Each phase consists of a number of activities. For example, during the concept phase stakeholders' needs are identified, development concepts identified, marketing and development of the future product are explored and viable solutions are proposed. The development phase comprises refinement of requirements, description of the solution and construction specification, verification and validation of the product. In the production phase the product is manufactured and certified for its operation. In the utilization phase the product is used, supported and maintained. Finally, during the retirement phase the product is stored, archived or disposed.

Software products have a slightly different lifecycle; for example 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 [Raj00] takes into consideration these characteristics, and defines the software lifecycle slightly different from the product lifecycle model (see

Figure **2**): 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 phase-out and closedown phase.

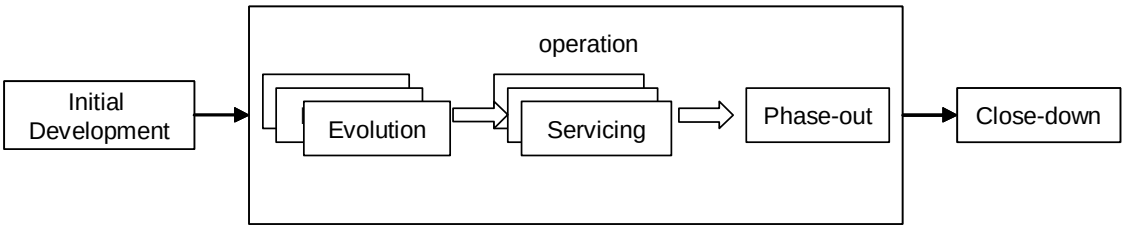

Figure 2: Software product lifecycle

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 the development organizations perform the same activities in the initial development phase as in each evolution cycle. Typically an existing software product will evolve into its next version by repeating the same sequence of phases, although probably with different emphasis. These activities grouped in define a *software development lifecycle* [KRU96].

Different models of the software development lifecycle have been proposed and exploit in software engineering [SOM04]. These models have shown strengthens and weakness in governing the activities that are required for a successful development and use of products. We can distinguish two main groups of models: Sequential and evolutionary. The sequential models define a sequence of activities in which one activity follow after a completion of the previous one. Evolutionary models allow performing of several activities in parallel without requirements on a stringent completion of one activity to be able to start with another one. In a sequential model phases and activities are the same or strongly correlated. In evolutionary models the phases are related to availability of the system to provide services (for example achieved through development iterations) and many activities are present in a particular phase. Well known examples of sequential models are waterfall model, or V model. Examples of evolutionary models, categorized as iterative and incremental development models, are spiral model, Rational Unified Process model, or different agile models. For more detailed descriptions of these models, their advantages and disadvantages see [SOM04].

Not all software development lifecycle models are suitable for all types of software systems. Usually large systems which include many stakeholders and which development lasts a long period prefer using sequential models. The systems which use new technologies, which are smaller, and to which the time-to market is important, usually explore evolutionary models which are more flexible and which can show some results much earlier than sequential models.

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

7.3 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 the development process**. 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 will appear. 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 will be 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, and for the component-level process, design for reuse will be the main concern.

Let us discuss these differences in more detail. To illustrate the specifics of the component-based development processes we shall use the Waterfall model - the simplest one – but the illustration can be relatively simply be applied for other development processes. Figure 3 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. The implementation phase will include less coding (in an ideal case no coding) for implementing functions, but selecting the available components, and if necessary adapting them 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 an ideal case that includes a full integration tool support, the glue code is generating automatically.

Figure 3 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 the components and component-development process (actually, there might be many parallel component development processes). These processes are depicted on Figure 4.

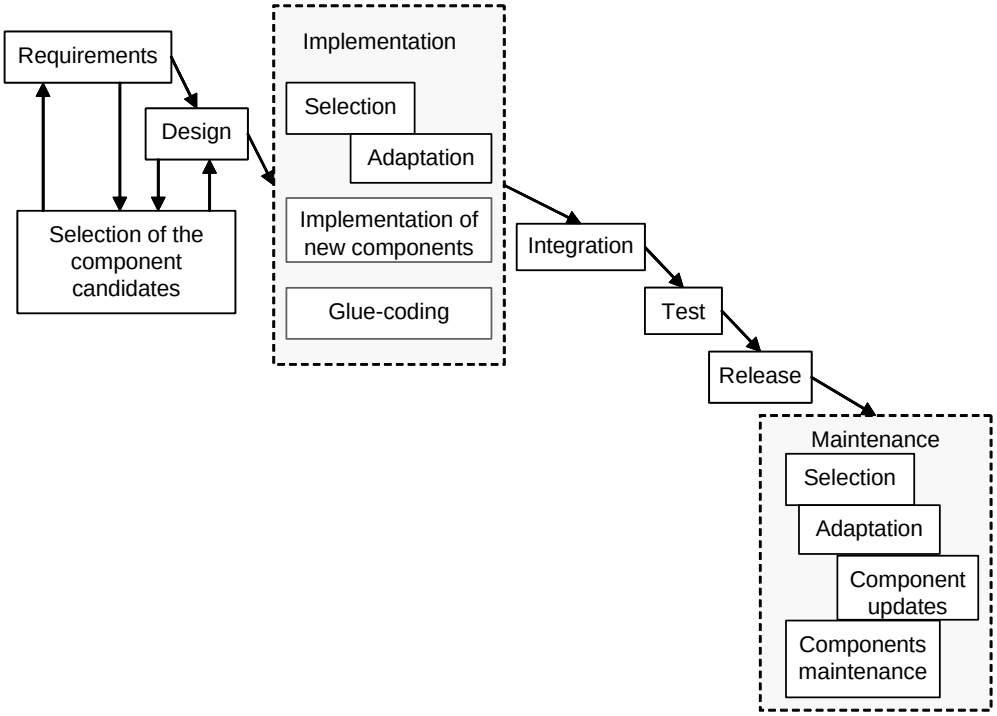

Figure 3: Component-based Waterfall Software product lifecycle

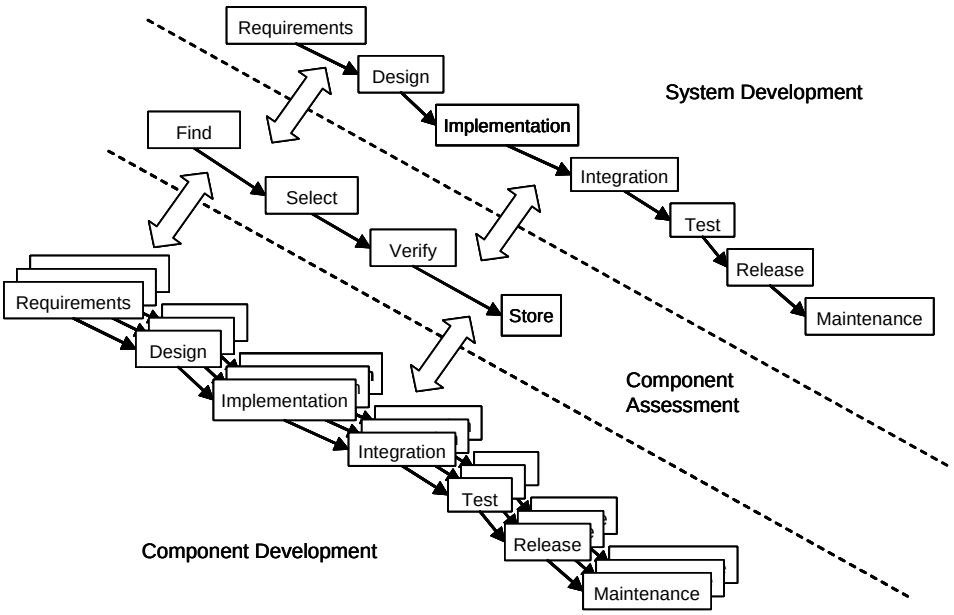

Figure 4: Parallel processes of component-based development

The processes shown on Figure 4 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 exist – these are similar decisions as staring points of the component assessments which originate 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, are related to the systems 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 7.4.

First, we shall discuss the activities of each process, and some of these activities in will be presented in more details the following chapters.

7.3.1 Component-based system development process

The main objective of the component-based system development process is the of system construction from (existing) components. This basic characteristic has impact on all phases of the development.

Requirements Phase

In this phase the requirements are collected, elicited, analysed and specified. 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 of availability of the existing components. This approach can be compared with obtaining a suit by order from a tailor who will make the suit according to our wish, or by buying a suit from a shop. In the second case we could not get any suit we wish, but take one available that suits most to our wishes. In the same way 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. More details about component-based requirements you can find in chapter (Requirements management).

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 necessary 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 processing 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 seen in Figure 3, the implementation activities only partially consist of coding – actually the more pure 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 however 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 would require 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 [Wal02] 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 integrated into the system.

The adaptation of components may be required 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:

- **Parameterized Interface**. Parameterized interface makes it possible to change the component properties by specifying parameters that are the parts of the component interface. These parameters can be used in different phases of the component life-cycle, depending on the component model – it can be a building parameter, or a deployment parameter or an execution parameter. An example of such parameter is a memory allocation, or frequency of execution, or a number of input data to be received in a row, or similar.
- **Wrapper**. A wrapper is a special type of a glue-code that encapsulates a component and provides a new interface that either restrict or extend the original interface, or to add or ensure particular properties.
- **Adapter**. An adapter is a glue code that modifies ('adapts') the component interface to make it compatible with the interface of another component. The intention of an adapter is not to hide or modify the component properties, but to adjust the interfaces.

Integration Phase

In a non-component-based development process the integration phase includes activities that build the systems from the incoming parts.[1] The integration phase does not include "creative" activities in the sense of creating new functions by production of new code, and for this reason there is requirement to automate and rationalise 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 standardises 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

[1] In some literature these parts as named as components. Since such "components" to not comply with our definitions of components (i.e. they do not confirm to a component model, we are not referring to them as components.

properties (in particular emerging properties, i.e. properties that are not visible on component level, but exist on the system level), remains complex and in many cases as difficult as for non-component-based systems.

Since system functions are not exclusively realised by components alone but often by a set of components, to verify these functions the components must be integrated before the entire system is built. For this reason the integration phase for component-based systems development process is spreading to earlier phases: implementation, design and even in the requirements phase. Fortunately in most component-based technologies the component integration is supported by tools, which makes the integration process simpler and more efficient.

Test Phase

During the test phase the system is being verified against the system specification (including both functional and extra-functional properties). In the waterfall model the test is performed after the system integrations, but this practice has exhibited many disadvantages. The more realistic is modified Waterfall model in which the test is performed for software units (such a variant is called V model). In CBD a need for component verification is apparent since the system developers do not necessary have a control on the 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 than performing in another environment [Wal02]. The component test can actually be performed many times – by assessment, when integrated in an assembly that provides a particular function, and when deployed (integrated) into the systems.

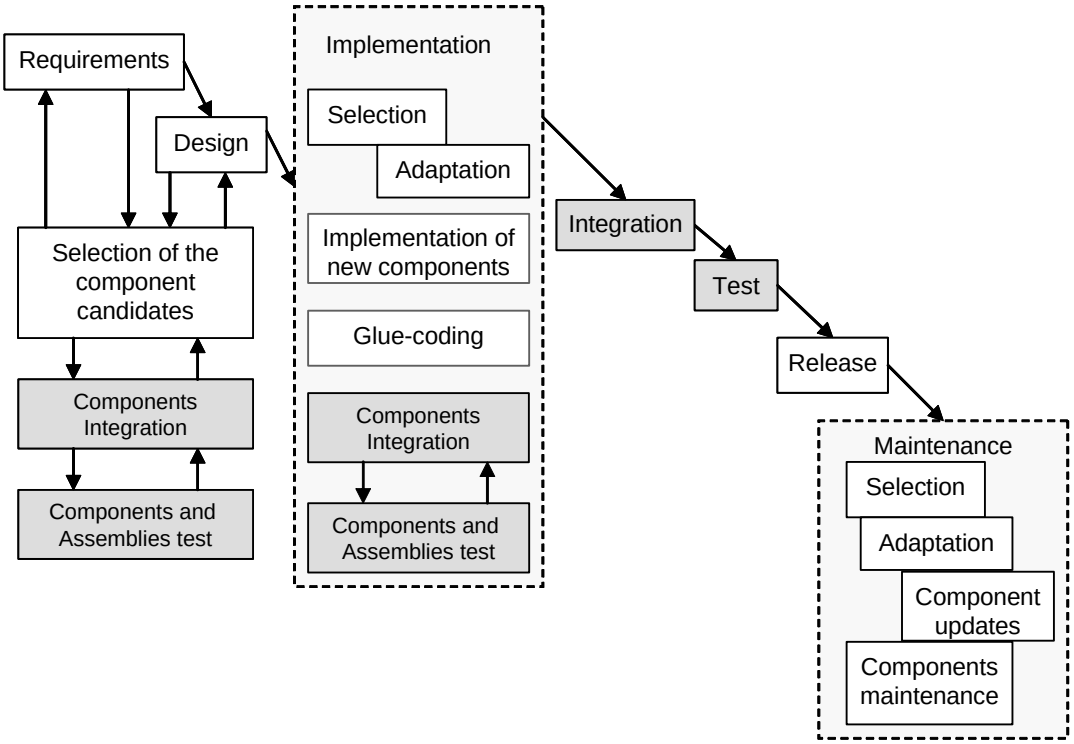

Figure 5: Integration and test in several phases of the CBD process

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 support by replacing malfunctioning component. The objective of 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 change. In principle there should be no need for their change. However the experience shows the opposite: The well known [MML96] law 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 system runs in. Even if the system functions properly, as time goes it has to be maintained. The approach of CBD is to provide maintenance by replacing old components by new components or my 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 5).

7.3.2 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, the developers have to find components that provide such functionality. Further they must verify that the selected components i) indeed provide the desired (or almost 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** – From an "infinite" component space find 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** – Select is a refinement of the finding procedure. Between the components candidates found, select a component that is most suitable for given requirements and constraints.
- **Verify** – Inevitable part of the component selection is the component verification. The first level of verification includes testing functional and certain extra-functional properties of a component in isolation. A second develop 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 should be stored in a component repository. The repository will include not only the component itself, but also additional specification (metadata) that can be useful in further exploitation o the component. Example of such data is measured results of component performance, known problems, response time, the tests and tests results and similar

These activities in the component assessment process are not necessary performed in the order as shown on Figure 5. Also depending on different architectural approaches (see section 7.4) 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" and "store" activity will not be necessary as the components would be stored in internal repositories.

7.3.2 Component development process

The component development process is in many respects similar to system development; requirements must be captured, analysed and defined, the component must be designed, implemented, verified, validated and delivered. When building a new component the 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 built to be part of some systems, preferably many. The components are intended for reuse in 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 or even they do not exist. 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 even not yet planned, the component designer many assumptions about the system must be taken. Many assumptions and constraints will be determined a selected component technology, for example component interactions, certain solutions built in the technology, assumptions of the system resources and similar. For this reason, the most likely is that at that the design time (if not already in the earlier phases) 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 a particular requirement in an efficient way. This requires much more design and development effort. According to some experience, developing a reusable component requires three to four times more resources than developing a component which serves a particular purpose [Szy98].

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 data transaction, database management, security, or interoperability support for distributed

software systems provided by component technologies .NET, J2EE, or COM+. Object-oriented languages are suitable for implementation of components since they use similar concept as CBD does, and since they contain elements that can be efficiently used for implementation of components. Examples of these elements are Interface in Java or virtual classes in C++, directly applicable for specification of component interfaces.

Integration Phase

While the component-development process does not include system integration activity, it is built to be easy integrated into a system. For this reason integration consideration must be in focus. An integration of component, if it includes other components, is also possible. 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. Not specific conditions should be taken fro grated, but the 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 the system developers.

Release Phase

Release and delivery of the components are inevitable part of the component development process. The components or assemblies of components are packaged into packages suitable for distribution and installation. The package will not only include the executable components, but also additional information and assets (specifications of different properties, additional documentation, test procedures and test results, etc.).

Maintenance Phase

The specific part of maintenance is a 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 responsibility if the component producer, or the system producer? Is it supposed that the component producers have obligation to fix the bugs and support its update in (possibly) the 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 infestation 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 get impression that an answer

from Component C was as delivered. The first analysis shows that he problem is in the component C, then B, then A, and finally input to A. The questions is who is providing that analysis if there 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 for many different reasons.

These examples show that maintenance activities can be much more extensive that expected. For this reason it is important that the component producers have build up a strategy how to perform the maintenance and take corresponding action to ensure the realisation 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.

7.4 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 [Crn03], 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 because of extensive support of component technology in modelling and specification, in easier implementations, in getting already existing serviced provide from the 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 are (shown on Figure 4) are reduced to two semi-parallel processes – system development and component development (see figure 6).

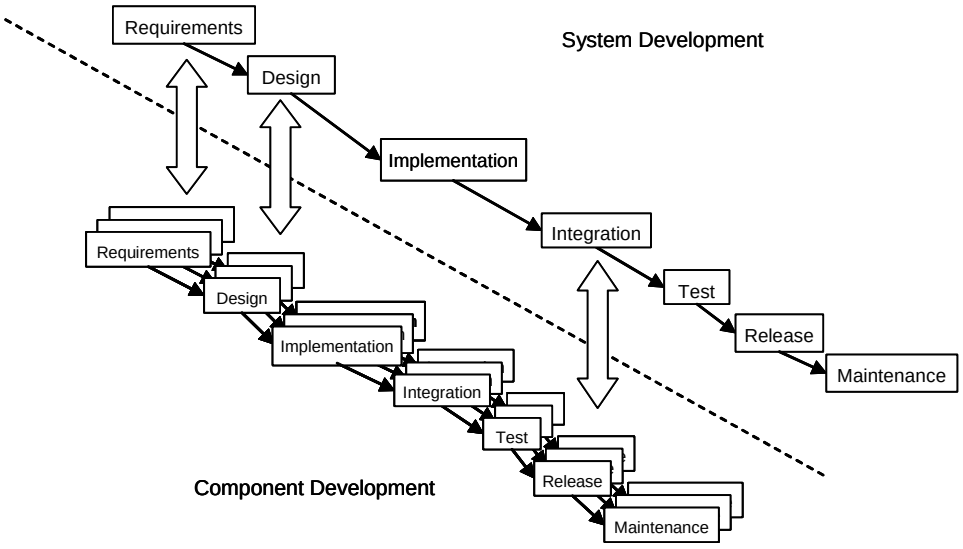

Figure 6. Architecture-driven component development process

Product-line development, which goal is to enable efficient development of many variants of product, or family of products 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 [COPA]. They are heavily architecture-driven, as the architectural solution should provide the most important characteristics of the systems. Within a given architecture (so called reference architecture) component-based approach places 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 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 best to the specified architecture. A second characteristic of product-line architecture (as a result of the time-to-market requirement) is parallelism of component development process and product development process and a combination of a top-down and bottom-up procedures. Referring to figure 5 we can see that all three processes (system development, component assessment and component development) exist, but somewhat changed.

COTS-based development, assumes component development process completely separately developed from system development. The strongest concern is time.-to-market from the component user point of view, and reusability from the component developer point of view. While COTS approach gives and 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 of the components is not specified and if different properties of the components are not properly and adequately documented. For the COTS-based development the component assessment plays a much more important role than in the previous two approaches. Figure 5 represents the COTS-based development best of al three approaches discussed here.

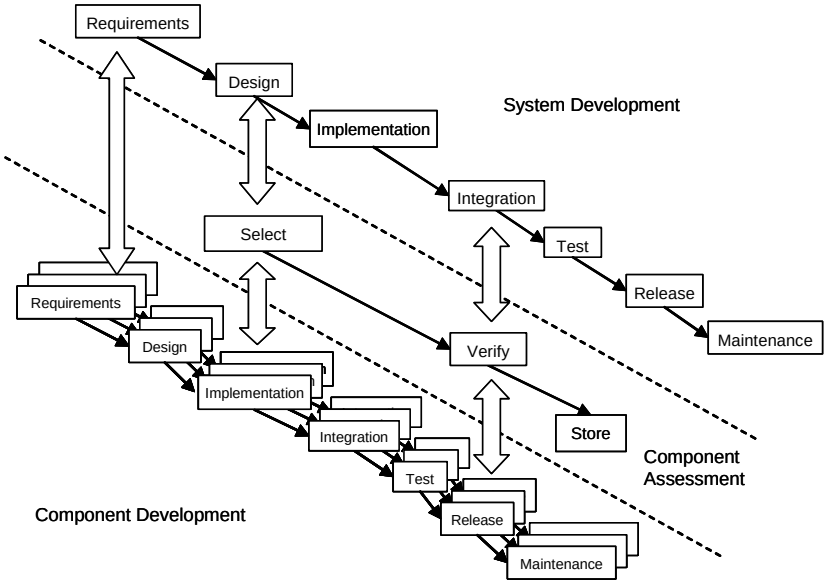

Figure 7. Product-line development

Which of these approaches are best, or most CBD-specific? There is no unique answer. While COTS-based development looks like the most inherited to CBD approach, and by this the most promising, the practice in last five-six years has not shown a big success; on the opposite, after a string enthusiasm on the market (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, and at the same time effective, simple and at the same time provide attractive functionality. Further a problem of trustworthiness (who can guarantee that the component is correct?), component verification and certification is not yet sold. Product line approach has been successful in many domains and in a combination with CBD-approach is a promising approach. Possible threat are increasing costs for development and maintenance of the component technologies developed internally, and that include compilers, debuggers, and in general 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.

7.5 Case study in product-line development

To illustrate a product-line architecture process let us look 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 component-based development using product-line architecture. The component model is internally developed and most of the tools are internally developed. The reason for that are the specific requirements of the 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 a hierarchical compositions: a composite component is treated as a standard component and in can further be integrated in another component. The 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 usually are different for different product variants. See Figure 8. Complementary to this horizontal layering there is a vertical structuring in form of subsystems. Subsystems are also related to the organizational structures; they are responsible for development and maintenance of particular components.

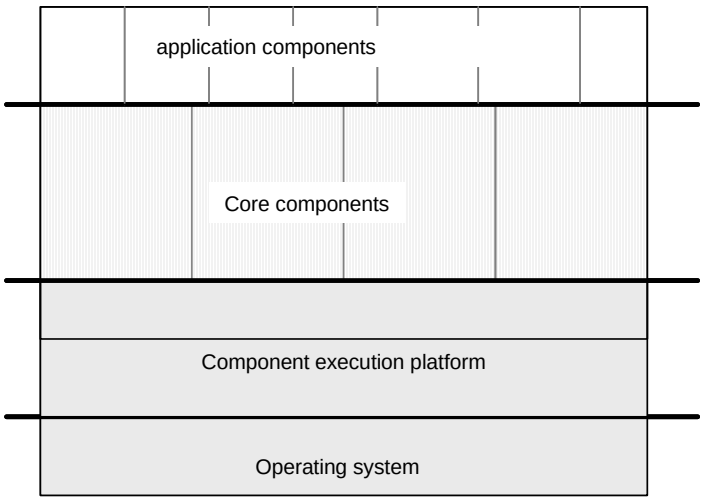

Figure 8. Product Software Architecture

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 moths. 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 9. The development processes in our case is manly 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.

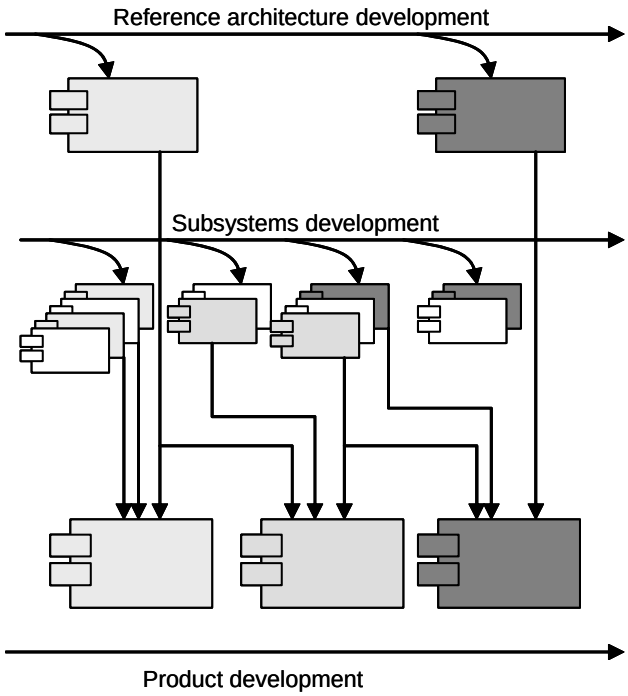

Figure 9. Products and components development processes

Although the overall development and production is successful, the company meets 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. All these problems point out that it is difficult to perform the processes completely independently; negotiation between different subsystems and agreements in many technical details between different teams are necessary. For these reasons coordination is necessary between development projects developing components and products. This reflects to the project and company organization. Figure 10 the overall organization of the projects. The following stakeholders have a special role 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 provide with the designs of their subsystems and coordinate the design decisions with other subsystems.
- Each subsystem has a test team and a QA manager which responsibility is the quality of the delivered subsystem components.
- The integration team which work on the delivery projects is represented by a product architect 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 of the component-based approach – more efforts must be put on overall architecture and test, and less on the implementation itself.

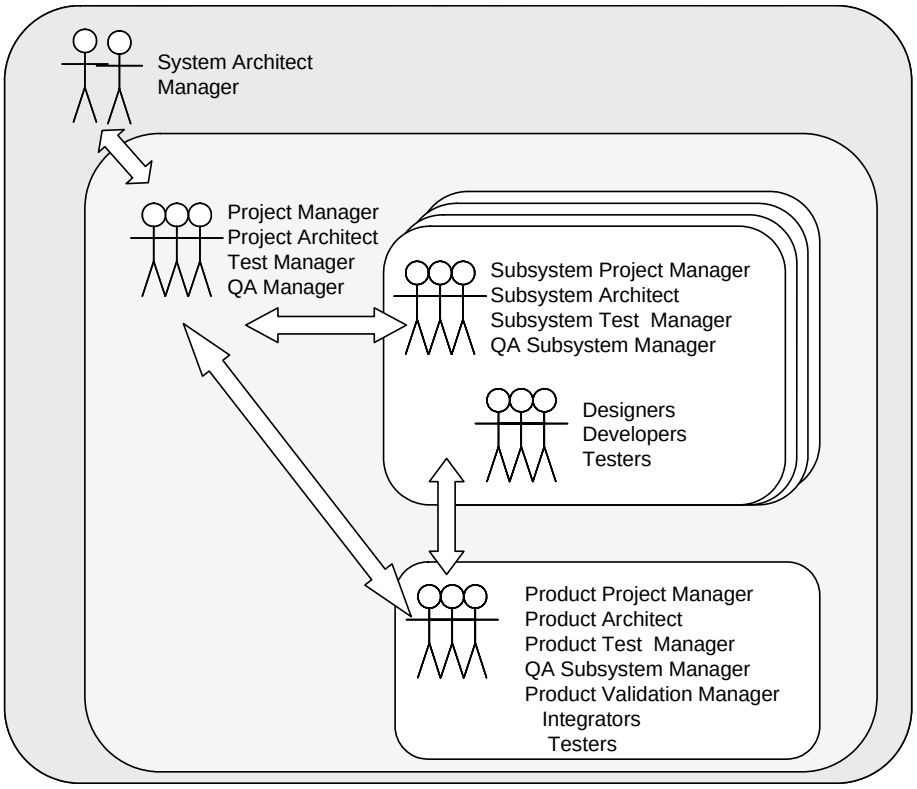

Figure 10. The overall project organization

7.6 Conclusion

In this chapter we have described different phases of component-based system life cycle. These phases are described in a frame of a particular process model, but similar principles are valid for any other development processes. The main characteristic of component-base development process 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.

7.7 Questions & Assignments

7.7.1 Questions

- Component-based approach decreases lead development time of systems, and enables shorter time to market. Does is however decreases overall efforts "embedded" into a products?
- Which activities require more efforts, and which less in a component-based development? Are these efforts equally distributed for development of systems and development of components?
- What are the main differences in development of components in product-line architecture, from COTS-based development.

7.7.2 Assignments

Illustrate the development processes when using another development model: Example

a) V model
b) Incremental model
c) an agile approach.

Describe the processes separately for system development and for components development.

References

[ISO02] ISO/IEC 15288, System Engineering - System Life Cycle Processes, First Edition, ISO/IEC, 2002

[KRU96] Kruchten, Philippe. *A Rational Development Process*, Crosstalk, July 1996

[Crn2003] I. Crnkovic and M. Larsson, CBSE – Building Reliable Component-based Systems, XXX, Artech House, 2003

[MML96] 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

[Raj00] Rajlich, Bennett. *A Staged Model for the Software Life Cycle*. IEEE Computer, July 2000

[SOM04] Ian Sommerville, Software Engineering, 7th Edition, Addison Wesley; May 10, 2004

[Szy98] Szyperski C., *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[Wal02] 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