Component-Based Software Development Life Cycles

Benneth Christiansson[1], Lars Jakobsson[2]

[1]Karlstad University, Division for Information Technology, 651 88 Karlstad, Sweden,
Benneth.Christiansson@kau.se

[2] Karlstad University, Division for Information Technology, 651 88 Karlstad, Sweden,
Lars.Jakobsson@kau.se

Abstract: Components are no "Silver Bullets" that will solve the problem of making software correct and robust. In order to make components more stable and correct the software industry needs to adapt a maturer life cycle model for developing, administrating and maintaining components. In this paper we propose a draft for how that could be achieved. Much can be learned from the life cycle model for application packages. The life cycle for a software system containing components must take into account that the system itself has a life cycle, as well as the component and all the other software in the system. Therefore we propose a merge of these life cycles in order to keep a system functioning and justifiable. Proposed in this paper is an outline for how the life cycle could be designed.

1 Introduction

Today, the need for well functioning and economically justifiable information systems is crystal clear. The quality of a company's information systems has become recognized as a strategic corporate advantage. Information systems of today can be viewed as the backbone of the modern enterprise and as such crucial to its functioning. The software industry is challenged by the new claims made by its customers. Sutherland [1] describes the situation today as:

"The global market has become an intensely competitive environment moving at an accelerating rate of change. To gain the strategic advantages of speed and flexibility, corporations must remodel their business processes, then rapidly translate that model into software implementations."

This means that the need to reduce time to market for software products is extremely crucial in today's software industry.

Schach [2] declares that:

"Software is more complex than any other construct made by human beings. Even hardware is almost trivial compared to software"

This quotation illustrates a fact that we think most of the actors in the software industry agree upon, constructing software is a very complex task in itself. At the same speed as hardware capacity is increasing, the complexity of the software solutions is increasing. This means that it is only getting more difficult to create the software solutions that today's customer require, because of the increased complexity in the required solutions.

In the increasingly competitive software industry the need for new innovative techniques to deliver satisfying solutions is greater than ever. This may in some sense explain the great belief and adaptation of so-called "Silver Bullets" in the software industry. A "Silver Bullet" is a new technique or tool that is made out to be the great solution to the different problems that the

software industry is facing [3]. The newest trend is considered something to adapt and use, new trends and tools are presented at an accelerating speed. Maybe the requirement for reduced time to market is not the only reason to this eagerness to adapt ”Silver Bullets” Jacobson et al [4] identifies a list of reasons:

”Increasingly, software organizations are facing simultaneous pressures:

- to reduce time to market;
- to reduce the cost of the product;
- to improve the productivity of the organization;
- to increase the reliability of the product; and
- to increase the quality of the product.”

The eagerness to adapt “Silver Bullets” has created a golden opportunity for people with good ideas to make big money coming up with new innovative techniques for software organizations to increase their competitiveness. We argue, in general, for a more well thought out and stable foundation for software organization’s to improve their productivity and in this paper we focus in particular on the life cycles of information systems. To us it is obvious that the new trends in the software community oftentimes lack the basic understanding for the fundamental problems concerning software development. Christiansson [5] defines the fundamental problem concerning software development as:

”To try to understand the customers sometimes unspoken needs/requirements and translate these into a tangible software solution.” (translated from Swedish)

This fundamental problem is addressed and handled by few, if any, new techniques in the software development community. We think this is one of the reasons to why we are still waiting for the ”Silver Bullet” concerning software development. Andersen defines an information system as a system for gathering, processing, storing and transferring information [6]. This system can be performed in part or as a whole by computers. The part of the information systems that is being performed by computers we choose to call software system. Langefors [7] described a software system as constituted by two parts the informal- and the formal part. The informal part contains issues such as required information and how the software solution addresses user-issues. The formal part contains issues such as data storage and computer instructions. He [7] claimed that the only way to create working software solutions is to consider both these sides. An interesting question is how many of today’s techniques for software development does that, if any? In a recent study it was concluded that 80% of the studied software development projects did not result in a, for the customer, satisfying solution. The most common reasons for this was exceeded budget or time to market [8] [9]. This would indicate that our statements about the fundamental problems are correct. Cox [10] illustrates this in a very graphic way:

”Software is a hybrid, halfway between an abstract idea and a physical tangible thing. Software is neither land nor sea, but swamp; a hybrid too thin for the army (software engineering) and too thick for the navy (computer science).”

In this paper we have chosen to focus on the need for maturer life cycle models in component-based software development. To be able to justify and argue for this need we will define and discuss important concepts and terms that we use in association with component-based software development.

2 Traditional software development

Traditional software development can be described by two extremes [11]. On one side we have the custom construction of a tailored solution fitting the one customer’s exact needs. On the

other side we have the development of application packages that is made from general demands with an entire customer-segment in focus. The custom-made approach has the advantage that the software system can support the customer's way of making business, if this is a unique way the customer can get the competitor-edge [12]. A drawback concerning custom made software systems is the cost for developing and time to market, the whole cost should be covered by the increased profit which using the system should result in. If several customers split this cost, as with developing application packages, the cost tends to be lower. Some other disadvantages with custom made software systems are the communication problem between the software engineer and the customer as mentioned above and the new systems' ability to communicate with other existing and yet to come software systems [11]. These disadvantages do not apply when acquiring application packages. There are however drawbacks with the use of application packages as well. One disadvantage is that when acquiring an application package one may need to reorganize ones way of making business to fit the application package [13] Another drawback is if competitors use the same application package that is not in itself a competitive edge. Yet another drawback is that when a company changes its way of making business it is very difficult to change an application package at the same time [13]. These two extremes should be considered as extremes and in fact many software development projects in reality uses a combination of these two extremes. The development of software systems should be imprinted with the use of situation-adaptation, which means that the development should be adapted to the present and unique situation i.e. there are no cookbooks for developing information systems. This adaptation can result in a combination of the two above described extremes.

2.1 The traditional life cycle models

When considering a software system, one can observe the system from a life cycle view. This means that the system is observed over time from the first notion of existence to the settlement of the system. Christiansson [14] makes a survey of several different software development life cycles presented by different authors. Christiansson [14] concludes that they have several notions in common and similar divisions of phases, or as they sometimes are called, processes. He describes these similarities in a general model of a software system's life cycle according to the following list (the right column consists of a short description of the purpose of each phase):

"Analysis	to understand the activities that the software system is meant to support,
design	to develop a detailed description of the software system,
implementation	to formalize the design in an executable way,
integration	to adjust the system to fit the existing software environment,
test	to identify and eliminate the nondesirable effects and errors and to verify the software system,
management	to keep the integrated software system up and running,
administration	to perform follow-ups of the management and perform consecutive revisions of the integrated software system, and
settlement	to settle the integrated software system (in part or as a whole) and to take charge of the information in the system." [14] (translated from Swedish)

3 Standard Application Packages and Components

In today's software development industry the trend is that more and more effort goes into building components, but little effort is made to ensure that these components have a life cycle matching the demands from the customers. There are no mature life cycle models for component-based information systems, neither from the user's view, nor from the developer's view. We believe that much can be learned from the research area of acquisition of application packages in general and the research concerning life cycle models for application packages in particular. Therefore we will in this chapter explain terms and concepts used in the field of standard application packages-research and the life cycle models presented there.

3.1 The Concept of Standard Application Packages

Anders G. Nilsson hands us one definition of a standard application package in his doctoral thesis "Acquisition of Application packages for Developing Business Activities – Development and Validation of the SIV method" [15]. According to his definition a standard application package must fulfill these statements:

- A standard application package is software that is already ready to use after minor adjustment to a specific organization's enterprise.
- The system must have been used previously in some other business.
- A standard application package is composed from one or more subsystems including applications and application data.
- Some subsystems must be pre-developed.

Further he states that a standard application package could be exemplified as

"a system containing support for the core product, services, information (and knowledge) and finance" [15].

Standard application packages can be composed in different ways e.g. they can be large integrated systems, but they can also be composed from smaller standardized modules. A definition for a standard application package, similar to the one discussed above, is:

"With a standard application package, we foremost mean such computerized information systems which either

- are acquired and used by several different customers and are developed and marketed by one supplier

or

- reused by one or more users and previously designed specifically for another user within or outside the enterprise itself." [16] (translated from Swedish)

The two definitions given above are both useful to us, not only defining the meaning of a standard application package, but also for components in the sense that a component as well as a standard application package are intended to be used by several users.

3.2 Lifecycle for Standard Application packages

The life cycle for a standard application package is actually composed using two different views according to Nilsson [15]. One life cycle can be identified for the user of the package and another life cycle is identified for the supplier of the package.

"A specific problems concerning standard application packages is that two different types of lifecycles can be identified: the supplier's and the user's view". [15] (Translated from Swedish)

From the customers' point of view, the lifecycle for a standard application package is very similar to the general lifecycle for an information system. The main difference is that the actual responsibility for the settlement of the application package is placed at the supplier part. The lifecycle from a customer view would look like this:

- Analysis of the enterprise
- Decision to acquire standard application package
- Adjustment of the application package to fit the organization
- Incorporation of the standard application package

As the reader will notice there is no phase for development, as the vendor of the package normally handles this phase. There is also no phase for administrating the application package. The administration and further development is the responsibility of the vendor, but the normal way of administering the package is to have a customer group with all the customers represented. This group will direct changes to be made in the future for the application package. The customer(s) will have to perform follow-ups on a regular basis in order to be able to tell the vendor what changes are preferred to the package. Nilsson [15] illustrates the lifecycle for a standard application package for the vendor and the customer, see figure 1.

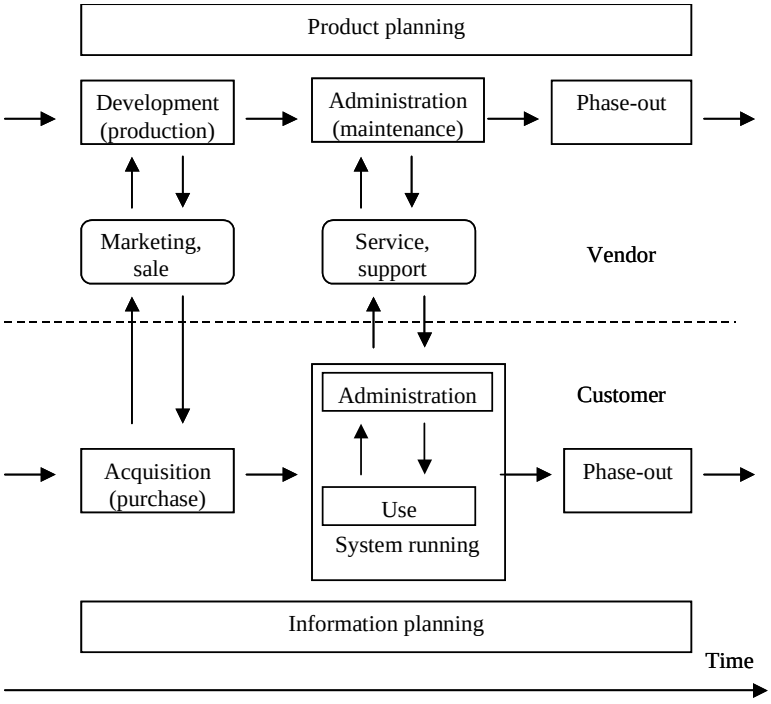

Figure 1. The life cycle for an application package

4 Component-based software development

One of the more recent techniques in the software development industry is component-based software development. This means that information systems are created through assembling more or less standardized software components into a unique software solution. The use of component-based software development can result in a way of software development that has all the advantages of the extremes described in chapter 2 and none of its disadvantages.

”The concept of component software represent a middle path that could solve this problem. Although each bought component is a standardized product, with all the advantages that brings, the process of component assembly allows the opportunity for significant customization.” [11]

We regard this statement as another ideal-situation or extreme, in reality a software development project should use whatever way to get the job done. This may mean an application package with a few added component or such.

The notion of software components is not new. Originally it was presented as early as 1968 on the NATO Conference on Software Engineering in a paper called ”Mass-Produced Software Components” by Douglas Mcllroy [17]. This paper is today famous and often referred to in component-based software development literature. He proposed a software industry that supplied off-the-shelf standard software components. He envisioned that programmers would combine these software components, instead of creating software solutions from scratch they would assemble software solutions from existing parts. We suspect that one of the reasons to why Douglas Mcllroy’s vision isn’t realized yet, is that each technique presented often only focus on the formal- or informal part of an information system. The predominance is on techniques that focus on the formal part. Some techniques focus on the informal part, they are very rare though and no techniques at all focus on both parts. Sims [18] illustrates this in what he describes as “Perpetuating the Great Mistake”. Every new technique in the software development community is either for, as Cox [10] so elegantly put it, the “Navy” or the “Army”, no technique, up to this day is for them both.

4.1 The software component

The term software component isn’t easy to define, it does not have a clear-cut definition in the software development community, but the meaning fluctuates. This paper does not focus on the issue of defining the term software component even though this is something needed and hopefully soon to be done. Instead we support the definition that Christiansson [5] makes. This definition is based on a survey and symbiosis of several more or less well-established definitions. We start our discussion concerning the meaning of the term with two quotations:

”A component is a unit of software of precise purpose sold to the application developer community…with the primary benefit of eliminating a majority of the programming the byer most perform to develop one or more function points…” [13]

”A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort [19]

These definitions, we believe, illustrates several of the more basic criteria such as a software components binary shape and its ability to connect without reconstruction to other software components. However these definitions does not include issues, such as identifying, maintaining and refining software components. Therefore we suggest a more mature definition of the concept software component. In this paper we will use a definition made by Christiansson [5], this quotation is translated from Swedish.

“A software component…:

- is independent and reusable,
- offers explicitly specified services through an explicitly specified interface,
- can affect/be affected by other software components,
- should have one documented specification (the software component described in a high level of abstraction),
- can have several independent implementation, i.e. one component can be implemented in several different programming languages, and

can have several executable (binary) shapes, i.e. one component can be executed in different software environments."

The fact that a component is independent and reusable shows that a component can be used without other components present, the services provided by the component should be accessible without any external help except from the software glue and necessary run-time environment. A component can affect and be affected by other software components. This means that two components can "work together" and "as a whole" create a greater service than used separately. In figure 2 we illustrate a software component with a context.

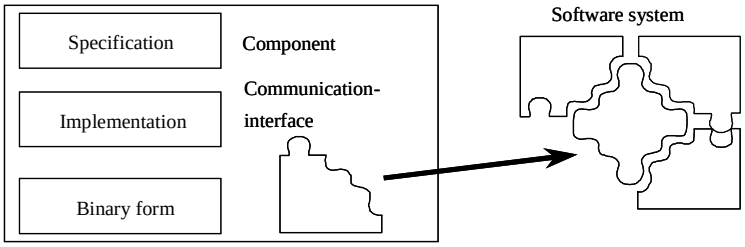

Figure 2. A software component with a context

The need for a documented specification for a software component is obvious if one consider the process of acquiring a component. How can one find a software component if one doesn't have something to look for. This is maybe the one single factor that can decrease the gap between the formal- and informal-part when developing a software system. If there are documented specifications, these can be described in such ways that they are useful when dealing with the development of the informal part of a software system and then the component as such is directly applicable in the formal part [5]. This notion of documented specifications can be elaborated to incorporate the need for standardization concerning specifying software components. Vaughn [17] implies that:

"A standard approach to building and using components must be set and universally practiced if the software engineering community is to reap the benefits…of reusable software components…"

With this quotation we want to stress the fact that standardization of software component specifications will be one of the next issues for solving the fundamental problem with software development [5].

4.2 Software components and reuse

A key reason for the recent hype concerning component-based software development is the possibility to reuse. Analogously with the blurred meaning of the term software component there is a blur concerning the meaning of reuse. Jacobsson et al [20] defines reuse as

"When we speak of reuse in software engineering we mean everything that can be reused at a later time."

This is a wide definition of the term and not very useful for understanding the notion of reuse. A more elaborate definition is made by Goldberg & Rubin [3]:

"Consumption of already completed artifacts is a strategy for completing a task with as little effort as possible."

In their definition they focus on the possibility to consume already completed artifacts when creating new artifacts. An artifact is something tangible or intangible constructed by human beings. Applying this definition on software development would indicate that besides the actual software source-code, any analysis- and design-result could be reused in a later stage. This is a more intricate definition but still not enough for our notion of reuse. We believe that the above definitions do not make any distinction between use and reuse, which we believe is a common mistake in today's software community. The use of something should not be confused with the notion of reuse. Reuse, we claim, is when a software development company makes a clear and systematic plan for how systematic reuse of created solutions will be performed in later software development projects [21]. This can include the use of software components but does not have to. There are several issues for an organization to consider in order to incorporate reuse. They would need to have certain employees that are responsible for maintaining, refining and spreading knowledge about the reusable assets for them to truly become reusable. They would need to handle the NIH-syndrome (Not Invented Here) that implies that software engineers are very skeptical in using software solutions they themselves haven't constructed [21].

Besides these more organizational aspects concerning reuse, there are other issues to consider, for instance when building software components for reuse one will have to handle the increased complexity of the software. This is due to the more generalized functionality the component need to incorporate be able to fit several different solutions and situations. Someone needs to finance this incorporation of more functionality [21]. This financing is always made in advance, and payoff is always uncertain, there can be no certainty that a software component really will be reused in the future [21].

There are also several paradoxes concerning organization and software reuse. One is the fact that on one hand we have the businesses which are developing software, they are interested in reuse for the possibility to more efficiently create new software products. On the other hand we have the businesses that consume software products, they are interested in reuse to be able to buy fewer software products in the future. This implies that the meaning of reuse varies if you are a producer or consumer of software solutions [22]. Another paradox is the fact that for a business to achieve reuse they need to centralize their organization. A central part of the organization needs to gather and distribute the reusable assets in the entire organization. This is paradoxical when considering one of the corner-stone advantages with software components, namely that more decentralized development is possible.

4.3 The composition of a component-based information system

A component-based information system is a more complex phenomenon than a traditional software monolith-based information system. A component-based information system can be regarded as consisting of three different levels (we choose to use the term level to avoid any confusion with terms such as tier or layer) [14] [21]. The innermost level is the component-infrastructure i.e. the components themselves and the necessary glue-code to make them interoperational. The middle-level is the software infrastructure, or the application infrastructure i.e. the grouping of cooperating components into software applications. The outer-level is the information system infrastructure i.e. the information systems that the different applications supports or consists of. These levels are described in figure 3.

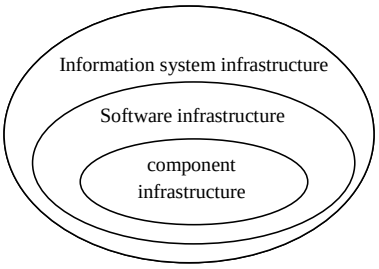

Figure 3. The composition of component-based information systems.

Each of the levels should be represented by an architecture that defines the infrastructure. This means that the existing infrastructure is due to the architecture that is applied. This implies that using an ad-hoc architecture would derive in an ad-hoc infrastructure. For each above described level there is a corresponding architecture, se figure 4.

“The architecture defines a systems components and their cooperation plus the basic structure and design.” [23]

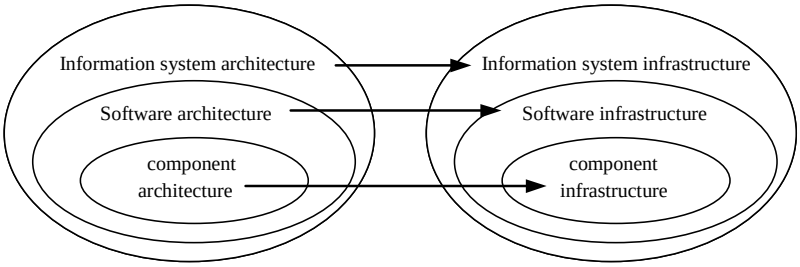

Figure 4. The different architectures and their correpsonding infrastructures in a component-based information system

In the software engineering community, architecture is not a novel concept. In this paper we choose to describe the three levels of architecture based on three quotations:

Information system architecture

“Information system architecture defines how a business distributes information and information-processing in different information systems and thereby demarcates them…” [24] (translated from Swedish)

Software architecture

”…the software architecture defines the static organization of software into subsystems interconnected through interfaces and defines at a significant level how nodes executing those software subsystems interact with each other.” [4]

Component architecture

“A component system architecture consists of a set of platform decisions; a set of component frameworks; and an interoperation design for the component frameworks.” [11]

A component framework as mentioned in the quotation above is a dedicated and focused architecture for a group of key mechanisms such as security and accessibility [11].

Szyperski [11] claims that there is only where these architectures are defined and maintained that the development and maintenance of components and component-based information systems is possible. This point of view is something that we find important to stress. We also claim that in the same fashion one need to define and use different life cycle models for the three different levels of a component-based information system (figure 5). It is not enough to use one life cycle model for a component-based information system, we believe that a more mature way of using life cycle models is to define new life cycles for each level in a component-based information system i.e.:

1. one life cycle for each component,
2. one life cycle for each application and consequently,
3. one life cycle for each information system.

By using three different life cycle models we obtain a more mature and nuanced description of the development and maintenance of component-based information systems. We can focus on a single component and/or an entire information system. We have models that describe their respective development and maintenance separately, and at the same time describe their dependencies. To illustrate, imagine an application consisting of several components, each component has to be acquired and verified before the application can be implemented.

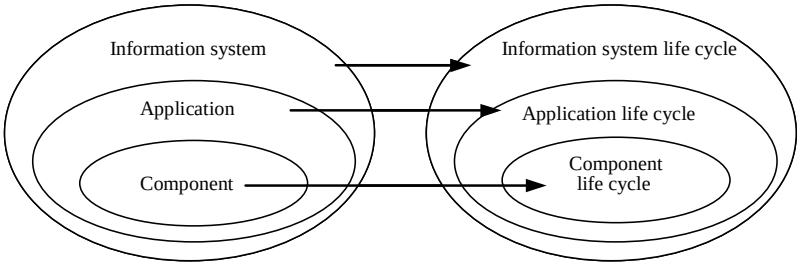

Figure 5. The three life cycles of a component-based information system

5 Component-based development – a double sided life cycle

The life cycles for component based development or for that matter for components differ in the sense that different actors experience disparate life cycles, depending on in which community they reside. The producers might have a perception of the component to be developed, as a product which life cycle is just like any other software. This idea could not be more wrong. A component is most of the time supposed to be an integrated part of a larger application or system, and because of that, questions that needs to be answered could amongst others be:

- What is the life cycle for the rest of the system?
- How will changes to a component in the system affect the behavior of another component?
- Will the component be compatible to newer versions of the surrounding systems and applications?

The customer on the other hand might perhaps have a different view of the life cycle for the component. From the user's view the component could be perceived as a stand-alone product to use as a plug-in for a larger application, which is purchased from a producer with whom no further contact will be necessary. That is fine as long as the customer does not upgrade the main application, but when the customer decide to upgrade the application using the component there is always the possibility that the component will not perform its specified task, or at least not properly. Using a component from one producer with an application or component from another producer will probably sooner or later imply an update of both producers' software. Nilsson [15] declares that:

"It is necessary that the customer and the producer society conduct some form of joint venture in order to co-ordinate the lifecycles of the different systems." (translated from Swedish)

5.1.1 Component-based development – the consumer perspective

The consumer of components may, as previously mentioned, perceive the life cycle of the component as very clearly defined. Firstly the customer acquires the component and then uses it within a software system of some sort. But there are several aspects needed to be addressed in order to maintain correctness of components, which are part of a larger system, both in regard to behavior and impact on the system as a whole. The component and its life cycle can not be viewed as a stand-alone product when used in a larger system. Therefore the life cycle for the component must merge with the life cycle of the system as a whole.

Most components today are sold in their binary form, which of course mean that the consumer of the component have difficulties making changes in the components behavior without aid from the vendor of the component. It can be argued that a component should be designed to have a certain behavior regardless of what changes are made to the surrounding environment. Unfortunately this would also imply that the component would have to be very robust, which in turn require the code itself to be designed to handle even incorrect use of the component.

The consumer of a system created by using components will have to address the problem that different producers may be involved in the development of the system. This should not produce problems for the consumer, but in today's software industry the components are not as well specified as for instance the hardware components. Until we have a very rigorous interface between components (as in the hardware industry), software component producers and consumers have to have a joint life cycle perception and cooperate, which would ease the consumers and producers maintenance of systems, applications and components.

5.1.2 Component-based development – the producer perspective

In the world of standard application packages, the life cycle for the producer would include development, maintenance and settlement. As we state in this paper the lifecycle for component-based software development is closely related to the life cycle of standard application packages, in the sense that components are mostly standardized packages of software in the same sense as standard application packages are. Therefore the need for a merge between the life cycle from a consumer perspective and the life cycle from a producer perspective is imperative to assert the correctness of the components in a system.

The producer life cycle will of course begin with the development of a component, which can be described in these steps:

- analysis of what needs the component is supposed to satisfy,
- design of the interfaces and component specification,
- implementation of the component,
- testing the component in different environments and situations,
- maintenance of the component, and
- settlement of the component or rather product replacement.

The result of this development phase will be a component, which satisfy the need for the present, but only the present. The producer must provide support and maintenance for the component, as the consumer can not alter the component to fit the consumer's needs. Furthermore the component must probably be updated to fit the consumer system when it is updated. This will of course demand a merge between the two life cycles. The maintenance of the component on behalf of the producer should be similar to the development phase.

The need to analyze what changes need to be made must be appreciated both from the producer side as well as the consumer side. This would represent the maintenance phase of the life-cycle model. Also included in this phase would be changes to be made to the component, initiated from the customer.

There will be no phase of system running from the producer's point of view, unless the producer is providing timeleasing, as it is the consumer who is running the component in his environment in most cases. However, there is a need of communication between producer and consumer in order to make appropriate changes to the component in order to fit the consumer's system. This communication could be represented as the arrows connecting the maintenance level between the vendor and the customer in the lifecycle model for standard application packages (figure 1), where the producer supply service and support to the customer within the frame of maintenance.

The customer does not initiate the settlement of a component. The producer is making that decision. Or rather, there are two different settlements, one for the customer, when he decides that the component no longer supplies the support it was intended for, or the component is no longer needed. The other settlement is the producer responsible for. The most likely reason for this settlement, from the producer's point of view, would be that a newer version or a completely new component would take over the task the older component handled, this is also called product replacement. This settlement, initiated by the producer, should be followed-up on behalf of the customer. The customer should have a fair chance to upgrade or at least have time to look for replacement for the old component (if it is at all necessary).

6 Outlines for the life cycles

In this paper our aperture has been to argue for maturer life cycle models concerning component-based software development and therefore we restrict ourselves to give rough outlines for the content of the different life cycles. We believe that this is an area for future research. We also choose not to describe the actual flow through the phases e.g. iterative-, sequential-, or incremental-workflow. Each phase is briefly described with only some propositional references to its actual content.

6.1 Information system life cycle

Analysis	Is the process of defining and understanding the activities that the information system is meant to support. Using and/or creating specifications for applications and/or components as pieces of puzzle can, for instance, do this.
Design	Is the process of developing detailed descriptions for the information system. This can be done by creating more detailed requirement-specifications for the identified applications and components.
Implementation	Is the process of formalizing the design in an executable way. Acquiring complete applications or components can do this. The acquiring can either be done through purchase, outsourcing, in-house development, component-leasing etc.
Integration	Is the process of adjusting the system to fit the existing information system architecture. This can include tasks such as adjusting components and applications to their specific software surroundings.

Test	Is the process of identifying and eliminating nondesirable effects and errors and to verify the information system. This can include both user-acceptance- and application integration-tests.
Management	Is the process of keeping the integrated information system up and running. This can include tasks such as upgrading and replacing applications and components in the information system.
Administration	Is the process of performing follow-ups of the management and perform consecutive revisions of the integrated information system.
Settlement	Is the process of settling the information system (in part or as a whole) and to safeguard the information in the system.

6.2 Application life cycle

Analysis/Design	Is the process of developing detailed descriptions for the application. Creating more detailed requirement-specifications for the identified parts i.e. aggregated components can, for instance, do this.
Implementation/ acquisition	Is the process of formalizing the design in an executable way. This can be done by acquiring the application or its aggregated components from a software vendor. The acquiring can either be done through purchase, outsourcing, in-house development, component-leasing etc.
Integration	Is the process of adjusting the application to fit the information systems it will be a part of. This can include tasks such as adjusting components to their specific environment.
Test	Is the process of identifying and eliminating nondesirable effects and errors and to verify the application. This should include tasks such as component-integration-tests i.e. the components are tested with focus on their cooperation rather than focusing their individual functionality.
Management/distribution	Is the process of keeping the application up and running. This can include tasks such as upgrading and replacing components in the application. If the application is a commercial product it should be merchandised and distributed.
Administration/Support	Is the process of performing follow-ups of the management and perform consecutive revisions of the application. If the application is a commercial product, administration will include tasks such as support and distribution of new versions and revisions.

Settlement Is the process of settling the application and to safeguard the information the application processes. If the application is a commercial product this may include tasks for product replacement.

6.3 Component life cycle

Analysis/Design	the same as described in previous section
Implementation/ acquisition	Is the process of formalizing the design in an executable way. This can, for instance, be done by acquiring the component from a software vendor. The acquiring can be done either through purchase, outsourcing, in-house development, component-leasing etc.
Integration	the same as described in previous section
Test	the same as described in previous section
Management/distribution	the same as described in previous section
Administration/Support	the same as described in previous section
Settlement	the same as described in previous section

7 Conclusions

Component-based software development is not a "Silver Bullet", as software components have not revolutionized software development. The fundamental problem concerning software development is defined as *to try to understand the customers sometimes unspoken needs/requirements and translate these into a tangible software solution.* Component-based development does not in our view offer a solution on how to understand the customer's sometimes unspoken needs/requirements. There is a need for standards concerning the way to specify software components. Maybe component-based development can aid when translating the customer's needs into tangible software solutions, through its new way of dividing the software into components. If component-based development is to be fruitful and beneficial there is still a need for development of the concept and content.

Traditional software development can be described with two ideal-types:

1) the development of a uniquely tailored information system, and
2) the purchase of an application package.

Developing a tailored information system is expensive and often financially hazardous, but if successfully performed could convey in competitive advantages. To purchase an application package is a more safe investment both due to the fact that it already exists and therefore is possible to evaluate and because the fact that more customers are sharing the developmental costs. A drawback is that it will not convey in competitive advantages due to the fact that the competitors can purchase the exact same application package. Component-based software development presents a middle path that entails the advantages of both extremes and lacks the disadvantages of the same. Component-based software development is the process of assembling software components into information system. This can be done through the use of standardized software components or the development of tailored unique components.

For a component to be well functioning in a software system, the need for the consumer's perspective on the life cycle for the component has to merge with the producer's view is imperative. This is due to the fact that component-based development must be regarded as a double-sided enterprise. The producer of components and the consumer of components must conduct a joint venture in order to insure system stability and correctness. A component can not be viewed as a stand-alone product, when used in a larger system, as this probably will lead to unexpected behavior from the system.

A software component:

- is independent and reusable,
- offers explicitly specified services through an explicitly specified interface,
- can affect/be affected by other software components,
- should have one documented specification (the software component described in a high level of abstraction),
- can have several independent implementation, i.e. one component can be implemented in several different programming languages, and
- can have several executable (binary) shapes, i.e. one component can be executed in different software environments."

Software reuse is an important issue when performing component-based software development. There is a lot of confusion regarding the meaning of reuse. Reuse in general can be defined as: *the consumption of already completed artifacts for completing a task with as little effort as possible*. This however does not distinguish between the notions of use and reuse. In this context reuse is when a software development company makes a clear and systematic plan for how systematic reuse of created solutions will be performed in later software development projects. There are paradoxes concerning software reuse. One of these paradoxes is the fact that on one hand we have the businesses which are developing software, they are interested in reuse for the possibility to more efficiently create new software products. On the other hand we have the businesses that consume software products, they are interested in reuse to be able to buy fewer software products in the future, this implies that the meaning of reuse varies if you are a producer or consumer of software solutions.

The traditional way of regarding a life cycle is to regard the information system as a monolith, and consequently there is one life cycle for the entire system. This is maybe a useful way when performing traditional software development, but it is surely not useful, but rather confusing when performing component-based development. This is due to the fact that a component-based information system by its very nature is not a monolith, but rather an intricate aggregation of parts i.e. components. The life cycle of a component-based information system should be described through the life cycles of all its aggregated parts.

There are three different levels in component-based information systems:

1) The information system level
2) The application level
3) The component level

These levels can be described as follows; an information system consists of applications that consists of components. We need different life cycles for each level. In this paper we present a rough and brief draft of how these life cycles can be described. We do not claim that this area is exhausted with our brief draft. Instead we mean that this rather is to be regarded as an expression for the need of mature life cycle models in component-based development.

The results presented in this paper are some of the results produced by the research project COMPASS (component-based information systems). COMPASS is a research project headed by Benneth Christiansson at Karlstad University, Division for Information Technology. The aim of the project is to identify effects in organizations when performing component-based software development. The project is a joint venture between Karlstad University, Linköping University and partners in the software industry.

8 References

[1] Sutherland J. (1996) *The Object Technology architecture: Business Objects for Corporate Information Systems.* OOPSLA'95 Workshop on Business Object Design and Implementation, Springer-Verlag, Berlin.

[2] Schach S. R. (1997) *Software Engineering with Java*. McGraw-Hill Companies Inc. New York USA.

[3] Goldberg A. & Rubin K S. (1995*) Succeeding with Objects, Decision Frameworks for Project Management.* Addison-Wesley Publishing Company, California, Menlo Park.

[4] Jacobson I., Griss M. & Jonsson P. (1997). *Software Reuse Arcitecture, Process and Organization for Business Success*. Addison Wesley Longman, Inc., California, Menlo Park.

[5] Christiansson B. (1999) *Komponentbaserad systemutveckling – genväg eller senväg?*, in Nilsson, A. G. & Pettersson, J. S. (eds.) *Om metoder för systemutveckling i professionella organisationer - Karlstadskolans syn på informatikens roll i samhället*, Studentlitteratur, Lund, (in Swedish).

[6] Andersen E. S. (1994). *Systemutveckling - principer, metoder och tekniker.* Studentlitteratur, Lund, (in Swedish).

[7] Langefors B. (1995) *Essays on Infology*. Department of Information Systems, Studentlitteratur, Lund.

[8] Veryard R. (1998). *Making CBD effective in your organization.* (september) http://www.scipio.org.

[9] Computer Sweden (1997) *Många projekt lever på katastrofranden*, nr 29, http://domino.idg.se/cs, (in Swedish).

[10] Cox B. J. (1990) *Planning the Software Industrial Revolution*. IEEE Software magazine (November), http://www.virtualschool.ed/cox.

[11] Szyperski C. (1997). *Component Software Beyond Object-Oriented Programming*. Addison Wesley Longman, Inc., California, Menlo Park.

[12] Casanve C. (1995) *Business-Object Architectures and Standards.* Data Access Coorporation, USA, Miami.

[13] Steel J. (1996). *Component Technology Part I An IDC White Paper*. International Data Corporation, UK, London.

[14] Christiansson B. (1998) *Komponentbaserade informationssystem - arkitektur, livscykel och systemutvecklingsprocess.* Arbetsrapport 98:4, Högskolan i Kalstad, (in Swedish).

[15] Nilsson A. G. (1991) *Anskaffning av standardsystem för att utveckla verksamheter*. Doctoral Thesis, Handelshögskolan i Stockholm, (in Swedish).

[16] Anveskog L. & Nilsson A. & Nord I. (1984) *Att välja standardsystem*. Studentlitteratur, Lund, (in Swedish).

[17] Vaughn T. (1990) *Issues in Standards for Reusable Software Components*. (november, 1996) http://ruff.cs.umbc.edu.

[18] Sims O. (1996) *Perpetuating the Great Mistake*. http://www.sigs.com.

[19] Microsoft. (1996) *The Microsoft Object Technology Strategy: Component Software.* www.microsoft.com.

[20] Jacobson, I., Christerson, M., Jonsson, P., & Övergaard, G. (1992). *Object-Oriented Software Engineering. A Use Case Driven Approach*. Addison Wesley Longman, Inc., California, Menlo Park.

[21] Christiansson (2000) *Komponentbaserad systemutveckling i praktiken*. Licentiate thesis, Institutionen för datavetenskap, Linköpings universitet, (In Swedish, forthcoming).

[22] Hughes B. & Cotterell M. (1999) *Software Project Management*. McGraw-Hill Companies Inc. New York USA.

[23] Asker B. (1995) *Arkitektur och systembygge för programvara.* Sveriges verkstadsindustrier, Stockholm, (In Swedish).

[24] Axelsson K. & Goldkuhl G. (1998) *Strukturering av informationssystem – arkitekturstrategier i teori och praktik.* Studentlitteratur, Lund, (in Swedish).