# Service based software construction process

M.Zinn

Network Research Group, University of Plymouth, Plymouth, United Kingdom
e-mail: marcuszinn@gmx.de

## Abstract

A lot of software development procedures include methods of resolution, which correspond to several kinds of problems. Some of these procedures have properties, which open the possibility to combine different procedures to solve bigger problem areas. The development of these procedures heightens the problems and complexity for the developers, architects and customers.

The motivation of the author is the development of a method of resolution, which includes:

- the advantages of today 's well-known software developing features and
- a service oriented approach combined with
- a component and service oriented approach.

The objectives of this research are:

- to reduce the known problems of software development,
- increase software quality and
- decrease (development) software complexity.

To reach this objective, this paper shows a software construction process as an alternative. The bases of the process are a model driven development approach and a component based software development, using new defined and type based services to provide and use components. Basically seven state of the art software development procedures will be analysed and classified. The focus of analyse is set on the component based procedures and the advantages of all analysed procedures. The result is a definition of a *Service based Software Construction Process* (SSCP) and a scenario for future research tasks for the author, which are based on using *Model Driven Development* and *Component Driven Development* inside the SSCP.

The research, on which this paper is based, and the paper itself show that such a development procedure, with service specialisation, is possible and achieves the requirements.

## Keywords

Procedure Model, Services, Software Construction, Model Driven Development, Component Based Software Development, Software Development Procedure

## 1. Introduction – State of the art software development

After careful consideration of "state of the art" software development models, a multitude of popular models can be found. All of them have different orientations. A common classification of these methodologies is not available at the moment. But these models can be described by analyzing their objectives. The following section

shows the objectives of a selection of todays accepted software development methodologies:

- Feature Driven Development
- Model Driven Development
- Agile Software Development
- Adaptive Software Development
- Extreme Programming
- Component based Software Development
- Service oriented Software Development

*(Models are chosen from the articles of the journal "Upgrade" (edition 08/2003 "Software Engineering State of an Art", (CEPIS, 2003) and edition 10/2004 "Software Process Technology" (CEPIS, 2004)) and the Website of Martin Fowler "The New Methodology" (Fowler, 2005). They are exemplary for other accepted procedures.)*

*Feature Driven Development* (FDD) and *Model Driven Development* (MDSD) demonstrate different ways to collect (customers) requirements. These requirements can be verified more precisely and can then be (automatically) transformed into a software product. FDD focuses on the feature-requirement definition. Each feature must be specified and every loop of software development iteration builds a feature or a set of features. MDD builds the set of requirements by defining a domain specific language and use this language to describe all requirements. An additional set of transformation rules must be given to (automatically) transforming them into source code artefacts. According to literature, the complete software development process is not being described by these two methodologies. The software developers can decide for themselves, which kind of procedure they want to use inside of FDD or MDSD. (Derniame and Oquendo, 2004; Meimberg and Petrasch, 2006; Björkander, 2003)

Another scenario is shown by the *Agile Software Development* methodology. The aim is to improve the efficiency of a complete software development process. For that purpose, *Agile Software Development* sets priorities on the aims and the aims supporting services, principals and methodologies. These preferred values are compared to the standard "bureaucratic methods and approaches". There are no technical rules given by this methodology. All approaches and methodologies are arbitrary and are used only to reduce the "red tape". There are a lot of other methodologies, which have linked themselves to the *Agile Software Development*. (Derniame and Oquendo, 2004; Paulk, 2001; Highsmith and Cockburn, 2001)

*Adaptive Software Development* (ASD) for example gets close to the agile process and tracks the objectives (like extreme programming (XP) and FDD) to get results by using transient and fast realisable software development phases. (Derniame and Oquendo, 2004; Highsmith and Cockburn, 2001)

The aim of the last group of methodologies is to construct software by developing and combining software components. In this group the accepted methodology is

*Component Based Software Development* (CBSD) (Stojanović, 2005) shows a analyse of other component based software development approaches, especially the revised form: *Service Oriented Software Development* (SOSD). The difference between these CBSD and SOSD is that the SOSD amplifies the CBSD with the basics of grid computing and service oriented architecture. Both methodologies focus on the development of components and their interfaces by the use of conventional procedures of software development. CBSD and SOBD dictate no rules on how to create and test components. Regardless the chosen methodology, it is important that a development process is chosen, which allows the implementing and testing of interfaces. (Cechich and Piattini-Velthuis, 2003; Apperly *et al.* 2003)

| | **Positive attributes** | **Negative attributes** |
|---|---|---|
| **Feature Driven Development** | Motivation for developers<br>• Good Controlling<br>• Milestones can be understood by non-developers<br>• short time to market | • Missing long time controlling |
| **Model Driven Development** | • Independent from platforms and procedures<br>• Flexible in development<br>• Process oriented<br>• Domain specific languages for communication between all parties<br>• Tool based | • Creativity of developers can disappear<br>• Deficiency of UML2.0 are Missing standards for MDSD tools |
| **Agile Software Development** | • Incremental<br>• Cooperative<br>• Linear process<br>• Adaptive<br>• Meta procedure | • project leader must choose the right agile procedures for the specific tasks |
| **Adaptive Software Development** | • Incremental<br>• Iterative<br>• For complex projects<br>• Tool based | • without controlling the system can easily get out of control |
| **Extreme Programming** | • Cooperative<br>• Communication<br>• Short time iterations | • Developers can easily get into "keen competition" |
| **Component based Software Development** | • Tool based<br>• Adaptive | • Components often have to be adapted, before reuse<br>• 100% reuse is rare |
| **Service oriented Software Development** | • good architecture for component<br>• Good experiences with system interations | • To less technical implementations<br>• Difficulties with many of data to transport |

**Table 1 - Typical positive/negative attributes of common development procedures**

Companies often mix some of these methodologies to achieve an added value by letting their different aims complement each other. As an example the literature suggests to combine the exact requirement acquisitions of the MDSD with the variable procedures of the agile software development (Stojanović, 2005). By use of

this approach the resulting agile software development process includes as few as possible adaptation iterations of the given requirements. The typical advantages which are hoped for by the use of this combination is a better adaptability of the own software development process and a saving of resources.

Basically, the following statements can be made by considering the advantages and disadvantages of the given development procedures and the analyse results of component based development procedures of (Stojanović, 2005):

1. Today's software development procedure models are process oriented and depend on the use of tools.
2. The actual processes can be combined in different ways and can be used together to build a software development procedure. This attribute suggests that future software development methodologies have to be combinable with other methodologies. The combining of different software development procedures can be described as meta software development procedures.
3. Most development procedure models build upon conventional software development and define themselves as software development procedures or -processes.
4. Procedures must be flexible and adaptable to specific domains and problems.

In comparison of the shown software development procedures with these statements, one procedure differs from the rest: Component based construction procedures (for example CBSD and SOSD). Because of the objective is to assemble components to a software system or to a higher aggregated component this is called Software Construction Process (McConnel, 2005).

(Fettke, 2002) and (Stojanović, 2005) show that there are only a small number of possible procedures in this area. In addition, these procedures do not adapt the attributes of other methodologies (like agile or adaptive development). It seems that software construction procedures, like CBSD, do not have the ability to compete against other procedures, because specific attributes are missing. (Stojanović, 2005) discusses this problem and shows a new component and service based approach. Especially for construction processes in relation to future technologies, these attributes can be very important. The *European Union*, for example, increased the subsidies for research in *grid computing*. The ambition is to become the leading union in this technology sector (De Roure *et al.* 2006). Technologies like *grid computing* support service based technologies and procedures.

There is the question, whether a software construction process, which includes most advantages and attributes of today's software development methodologies, can be defined. Additional such a software construction process has to be based on a service approach.

The following sections will analyse *component-* and *service oriented software development* as an example of construction procedures. In addition, the concept of

*model driven development* will be explained, because the new process will be based on this approach. Afterwards a process will be illustrated, which represents the method of resolution for a new kind of construction process. At the end of the document a concrete scenario will be presented, which includes a description of the authors research task.

## 2. Software Construction Procedures and Model driven Development

The idea to construct software, which means to build software by combining different components, is not a new idea. (Szyperski, 1999; Fettke, 2002) The basis of software construction are components, because components will be fit together to build software or a system. The literature shows (Fettke, 2002) that there was no standard definition for the term "component" till a short time ago. Today's discussion and research groups got a standard definition. The *Gesellschaft für Informatik e.V.(GI)* published the following component definition:

"*A component consists of different kinds of software artefacts. It is reusable, complete, marketable, offers well-defined interfaces, hides its implementation and can be combined with other components which are not known at the developing time.*" (Ackermann and Fettke *et al*. 2002)

CBSD and SOSD are the today's preferred kind of software construction. To define a new software construction process, the basic rules and advantages of these procedures must be shown and analyzed.

### 2.1. CBSD

As shown before, the CBSD focuses on building large software systems by combining pre-built software components. CBSD embodies the "buy, don't build" philosophy. This means, if a component is missing, it can be delivered by another component publisher. (Brooks, 1987) Figure 1 shows the typical activities and states of the development of components and software construction in CBSD. To get a common understanding of component based software development, the activities will now be explained.

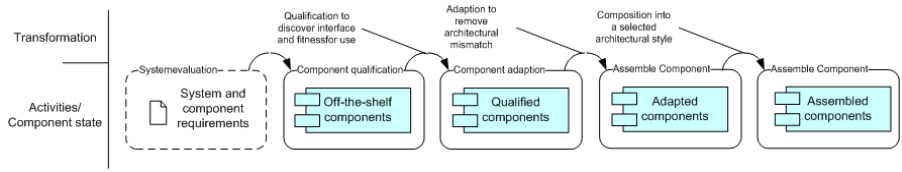


**Figure 1 – Activities and states of component software development according to (SEI, 2007)**

2.1.1. Component qualification (SEI, 2007)

In this activity, previously-developed components, whose properties fit the requirements, will be searched. These components are able to work in the new system context. Selecting a component can be very difficult. This activity can be divided into two phases: discovery and evaluation. The discovery phase includes the identification of the searched properties. There are also common properties, which are difficult to discover: reliability, predictability and usability. Additional "non-technical" properties have to be considered. For example vendor's markets share, past business performance, and process maturity of the component developer's organization. During the evaluating phase the discovered components have to be checked against the given requirements. The literature shows different standardised ways to discover and evaluate components by defining quality characteristics. See (ISO/IEC, 2001) and (Poston, 1992) for further information.

2.1.2. Component adaptation (SEI, 2007)

Because most of the time, parts of components do not fit exactly on the requirements, they have to be adapted with other components. To minimize the conflicts between components, some rules have to be considered (Valetto, 1995):
*"white box, where access to source code allows a component to be significantly rewritten to operate with other components",*
*"grey box, where source code of a component is not modified but the component provides its own extension language or application programming interface (API)",*
*"black box form of the component is available and there is no extension language or API".*

2.1.3. Assembling components into systems (SEI, 2007)

The next step in the CBSD activities is to assemble components. The basis for this component construction is an infrastructure, given by the architect. Typical infrastructure systems are: D*atabases, Blackboard, Message Bus, Object Request Broker (ORB), Common Object Request Broker Architecture (CORBA).* The result of this activity is a software system.

2.1.4. System evolution (SEI, 2007)

The written or evaluated components must be includable into the system. This step must be planned at the beginning of the project. This also includes several short- and long-term considerations, which have to be involved into the project plans. For example:

- ***Requirements:*** Usually pre-existing components will be used to assemble a software product. These pre-existing components were built with pre-existing, and possibly unknown sets of requirements. With general requirements at assembly time, these components have to be minimally changed for use, or not changed at all. Most of the time, a pre-existing component doesn't exactly match the given requirement. In the worst case,

the architect has the problem to find a new component or to change his system architecture.

- ***Reuse of existing components***. Most times the existence of components in an organisation is not known. If Component-based system development is used, reusable components are important. The reuse of existing components limits the needed resources (cost, time, man hours, etc.) for a project. Often, existing components have to be adapted.
- ***Architecture***. The selection of standards and components needs to have a sound architectural foundation, as this becomes the base for system evolution. This is especially important when migrating from a legacy system to a component-based system.

More examples can be found at (SEI, 2007).

## 2.2. Service oriented Software Development

The topic service oriented software development is a new one. The basic principles are the same as in CBSD, software can be built with components. But the complete topic is not well defined at the moment. At the current state of research SOSD is only an idea to transfer components over services. (Stojanović, 2005) for example shows that the idea is to use (web based) service and service oriented architecture to manage and provide such components. The important advantage of this kind of development is that SOA and service oriented computing (SOC) are a good way to build flexible systems at the moment. This is very important for new development procedures, because the components not only have to be replaceable but also the development procedure and the system in use must have the possibilities to do this.

This section shows that two attributes are very important. The first is the definition of components. As seen it is not easy to define what can be a component and how it can be built. But without the definition of a component there is no way for component based development. The second attribute is the way of how a component can be used and published. SOSD shows a way with accepted concepts and technologies.

## 2.3. About Model driven Development (Meimberg and Petrasch, 2006)

Compared to CBSD and SOSD the MDSD is not a typical kind of software construction approach. MDSD is a model driven procedure model, which concentrates on architecture. As seen in the introduction, the main objective is to find a way to collect requirements and transform them (most times automatically) into artefacts, which can be used in the development process. Basically the methodologies of *Product Line Engineering* and of the *agile software development* will be used. The basis of *MDSD* is the *Model driven Architecture (MDA)*. MDA is a concept which is placed between procedure models and development methodologies. It separates the recurrent artefacts (for example definition of platforms) from the project specific artefacts. It provides the needed artefacts of a development process and is not contradictory to software development methodologies. MDA needs the

context of a procedure model and must be adapted to the specific procedure model, the project itself and the project management.

(Meimberg and Petrasch, 2006) defines MDA**: "***MDA provides a systematic approach for the development of software systems. A minimum of quality risk and a maximum of automation will be created, because of a high formalism of the different models, which will be transformed iteratively. The architecture is closely connected with the perspectives and views, which considers the aspect of software systems comprehensively, to get few or none amendments after the transformation."*


**Figure 2 - General transformation process of MDA**

Figure 2 shows the general transformation process of MDA. MDSD additionally concentrates on:

- describing the complete system, which will be built in different models and on
- describing each source artefact of each model.

To do this a *domain specific language (DSL)* will be defined and understood by the domain specific customers and the development team (developer, architect and designer)

The results of this process are

- a graphical representation of the models and the source artefacts as a *Platform Independent Model* (PIM), based on the specific DSL,
- Transformation rules to transform,
- Source code as a *Platform Specific Model* based on a specific programming language (defined in the Transformation rules).

Usually the *UML 2.0* will be used to describe the *PIM*. The transformation rules must be developed after the definition of the specific *DSL*. Through these rule sets interfaces, methods, classes and code snippets will be created. (Bettin, 2004) describes this as "component specification".

# 3. Service based Software Construction Process (SSCP)

## 3.1. Overview

The previous chapter, shows the current situation of software construction. As seen, the point is to define a software construction process, which includes the typical attributes and advantages of today's accepted software development methodologies and is prepared for future technology bases. A new definition of a software

construction process is presented in the followings section. The central statement of this "new" process is that software can be built with components by using common procedures and up to date technologies. Components can be provided and used by services. To preserve the required attributes other procedures or methodologies can be included in this methodology. In addition, the developer's view of components and services has to be changed. This adapted view focuses on software development and software construction. Because of this, the role of the "software architect" gains the focus and becomes now more of a software designer.

At the current state of research the following required attributes have to be included in the new process (*Summarised from table 1, Column "Positive attributes" in the first chapter*):

- Process oriented
- Supporting Milestones
- Possibility of short and long time iterations
- Possibility of platform and procedure independency
- Supporting Developers
- Supporting Communication between all parties
- Incremental
- Cooperative
- Component based
- Service based
- Combinable

The next section shows the structure and tasks, the new construction process has to include when achieving the given requirements.

### 3.2. SSCP in detail

To achieve the requirement of combinability of other methodologies with the SSC, it is necessary to divide the SSC into different phases. A construction process as described should be consisting of the following three phases.

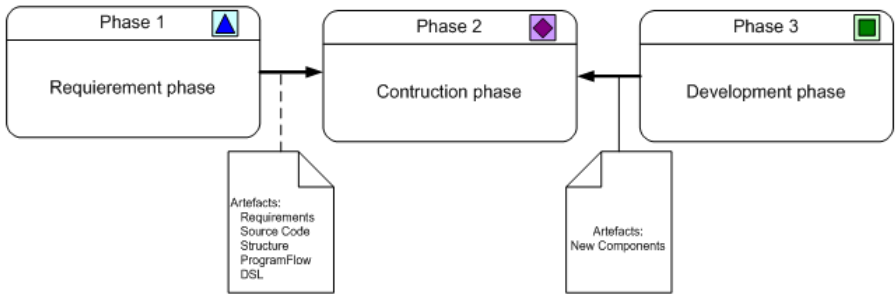


**Figure 3 – Phases and artefacts of the SSCP**

#### 3.2.1. Requirement phase

The procedure, used in the requirement phase is not given by the SSCP. Every procedure which automatically creates source code and a graphical representation (like MDSD) is qualified for this phase. Otherwise this information has to be built manually. All requirements for this phase are designed and collected in a domain specific language. In addition, all given information has to be verified manually or by provided tools. The results of this phase are two artefacts: (automatically) designed code and (automatically) designed graphical representation or problem description.

#### 3.2.2. Construction phase

The software construction phase is the core part of the SSCP. In this phase finished components will be assembled. The result is a software product or higher aggregated components. To work with a graphical representation is necessary at this time, since it accelerates the process and gives the process control. As shown before, the basics of CBSD / SOSD will be used. The software designer combines all needed components, the interface and the sequences (Sequence is equivalent to program flow, see section "*Type 4 Structure*" at the end of this chapter). These components will be published by services. Components which are not provided have to be developed by the developers themselves (see development phase).

#### 3.2.3. Development phase

All missing or insufficient components will be created, adapted and managed in this phase. The choice of the development procedure model has to be made by the project leader or the designer. The following methodologies are qualified: CBSD, miscellaneous kinds of agile procedures and SSCP.

### 3.3. What is new?

The overview about the "new "development procedure poses a question: What is new or innovative of the SSCP?

The answer to this question will become clear on closer inspection of the given phases. In the first phase there are no new innovations, known standards can be used. The third phase has the same attributes. At this point, software development will be utilised to build and publish new components. The choice of the development procedure is free. These two phases are defined to make the SSCP a complete process and to afford the combination of different methodologies in the SSCP. The process is ready for integration. The most interesting phase is the second one. At this point the software designer has to construct software.

At this state of research the architect has to follow these guidelines (*The guidelines are under research. At the moment they are deduced from CBSD/MDA rules.*):

1. The result of the first phase, Source Code and graphical representation, are the basics for the construction. By using this information, the designer can deduce components, classes and sequences. (see (Meimberg and Petrasch, 2006))
2. The designer is allowed to use components only to build the software!
3. The designer must be able to work with aggregated (system integration layer) and atomic (function calls) components.
4. Compliance with the guidelines of CBSD/MDA/MDSD.
5. Compliance with the definition of components and attributes - see (Ackermann and Fettke *et al.* 2002).

On closer inspection these guidelines are deduced from attribute requirements for the SSCP. To assist the designer in following these rules, special types of services will be defined.

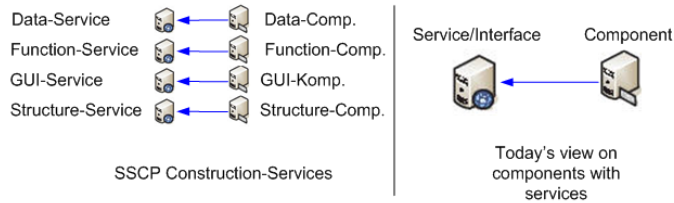


**Figure 4 – SSCP-Service types**

Each type of service will deliver special kinds of components. The definition of a component and of a software artefact will be expanded. Individual parts of software are also a kind of component. Figure 4 shows the four defined types of services components: Representation (GUI), Data, Functionality and Structure. Because of this, service types are for use in software construction. They are now called *Software Construction Services* (SCS). These services and their functional range have to be surveyed:

Type 1 – Representation: The first service type delivers one or more graphical user interfaces (GUI). The information, this service provides can be source code, binaries or the description of a GUI for local or remote components. Most of the older GUI technologies are hard coded in a specific programming language. Often there is no way to change GUI technology without high costs. New technologies for using GUIs are based on the *extensible mark-up languages* (*XML*). The objectives of these technologies are the description of high level GUI functions (like high colour depth, GUI logic, animation, 2D/3D, vector graphic etc.).The added value is a light weight GUI (like *HTML*) with the possibilities of a GUI used in graphical operating systems (like windows). A high level GUI language like this is qualified for a *GUI-SCS* because it only has a smidgen of data but high possibilities for the architect and the designer.

Type 2 – Data**:** The second service is the *Data-SCS*. Data implies all information which will be displayed, modified or used as the base of decisions. It is important that the Data-SCS is a data query. That means there is no functionality with high

costs for generating data (see the *Functionality-SCS*). Behind a *Data-SCS* there is only a data providing system. Data types can be simple types (for example integer, strings, boolean etc.) or complex types (dataset, picture etc.). To transport data types in a service most programming languages provide serialisation. This means self defined and normal data types are serializable. Thus, the platform independence is assured. For example, today's web services use xml to serialize data and data types. Data has to be stored permanently or temporarily. Each kind of software includes possibilities to store data. There are two possibilities of storing data in software: internal and external. Software developers use variables to store date temporarily. The Software constructer in *SSCS* also uses such a method for storing data. To store data externally, databases or storage systems will be used. For use by the software designer, a *Data-SCS* is integrated in *SSCP* for such tasks.

Type 3 – Functionality**:** The *Functionality-SCS* represents functionality. Usually software developers have to develop missing functionalities by themselves. When using *SSCP*, the functionality comes from a *Functionality-SCS*. If there is no *SCS*, the developer's team has to build a new component. Today, a service is defined as a remote executable functionality and can be used for example with *web services* and *web service definition language (WSDL)*. A *Functionality-SCS* can do the same things, but also includes the properties to provide complete components. These components can be executed local or remote. The disadvantage of local execution is the partial loss of platform independence. In some situations, on the other hand, some functions are so critical, that they have to run locally. In these cases a remote execution is not possible or has many disadvantages.

Type 4 – Structure: Structure of data, classes and methods is the content of the *Structure-SCS*. Theoretically the software architect designs different diagrams (for example in *UML*). These will be converted into source code. Some tasks of structure design are given to the software architect (for example interfaces) and some are given to the software developer (for example patterns).

In the case of *SSCP* the question of structure is passed to the software architect. By using atomic components (SCS-Components) to build software, the architect has to make the structure decisions. In the *SSCP* it is possible to use two kinds of structures. The first one is the outer structure. It will automatically be built as an artefact of the requirement phase of the *SSCP*. This code is based on the domain specific language defined by the software architect and the customer. The architect gets the general information about the sequence of the program. Additionally he gets the information about the first classes, packages and components needed for this software and the task he has to complete. The second structure is called the inner structure and is built by the architect himself or is provided by a service. There are two kinds of structure definitions:

1. Code structure (for example class descriptions, patterns)
2. Component structure (for example interfaces, dependencies)

This simple classification of components and services in four basic types does not fully correspond to the current view of components and software artefacts. Figure 5

shows the old and the new view to components. The new view has interesting effects on the construction process, the roles used by this procedure, and the results of the process. To provide type based services, current technologies have to be adapted. The initial question is, whether something new and innovative can be answered with yes.

The objective of the research is to find out if it is possible to create the SSCP like shown above. Another interesting question is how applicable and adaptable the new process will be in the "real world". To test this, a research scenario must be defined.

## 4. The SSCP research scenario description

The previous sections present the common structure of the *SSCP*. There are a lot of possible variations of using different software development procedures with the *SSCP*. To do a research into this kind of process, a concrete scenario has to be defined. The first iteration of this scenario will focus on the second phase, its implementation and usability. The other phases and the details of interaction have to be given by common standards or procedures.

The *SSCP* will basically use *Model Driven Development* and *Model Driven Architecture*. (*It is important for the author to use a common procedure, which is discussed in today's established journals and scientific papers*.)

MDA/MDSD also represents a well defined procedure. It uses and supports today's common technologies and concepts. These approaches are qualified for use, because they are based on known methodologies and concepts. MDA/MDSD has the ability to gather requirements and transform them into a model driven approach. The SSCP will use:

- the definitions of *MDA*. (Definition of artefacts, syntax, semantic etc.)
- the typical procedures and technologies, used in *MDA/MDSD*. (Domain specific languages, *UML*, etc.)

Figure 5 shows the scenario for the first research iteration with technologies, concepts and procedures.

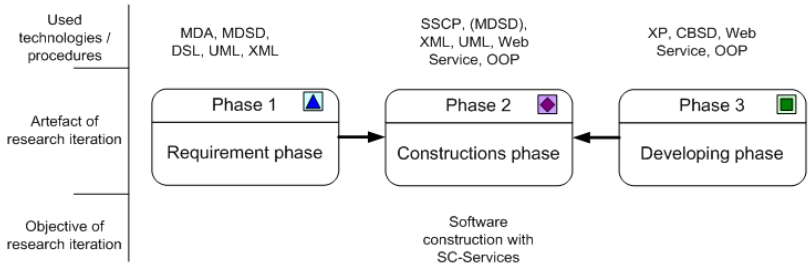


**Figure 5 - Research scenario overview**

In the second research iteration, the first and the third phase gain the focus in this scenario. At this point it is important to adapt the procedures of these phases to get a better support for the second phase, for example the use of the *Y-Approach, which also includes the testing phases,* instead of the *MDSD*. Additional research on the communication and the communicated artefacts between the phases is very important.

Evaluation phases are necessary before and after of the second research iteration. In these phases the SSCP has to be tested and proofed. There are three different kinds of evaluations: The first one is to compare the features of the SSCP with the features of other component based development procedures. The result is a theoretically comparison and a list of advantages and disadvantages. The second evaluation content is to proof the SSCP as an extension of the MDA/MDSD. It can be also necessary to compare SSCP with other MDA based procedures, which have a similar approach. SSCP must have some benefit for the MDA/MDSD. The third kind of evaluation are two practical tests. The first one is to proof the practical use of the SSCP and a known similar procedure in a fictive software development project. The second one has to be a "real world test". The second procedure of the first test has also be used in this scenario for comparison. To proof the SSCP an additional research task about evaluation of component based software development procedures has to be added to the overall research plan.

At the current point of research, standard technologies will be used, for example webservices for the *Software Construction Services* and XML based languages for the domain specific language. During the research, service providing technologies will be considered. At this point the new service provision approach of (Heckmann, 2007) will be proved.

## 5. Conclusion – Result of the current project state

This paper introduces a new way of software construction. This "new" software construction process consists of three phases (Requirement-, Construction- and Development phase) and contains most advantages of other common software development procedures.

Each of these three phases is based on different software development procedures, for example MDA or CBSD. The core of this construction process presents a new kind of view on components and construction. This view includes, that components and their artefacts will be provided and delivered by type based services. These types are *Data, GUI, Structure* and *Function*. By use of these services, software can be built and executed.

The current state of research shows, that an approach like this one is possible. Simultaneously this paper shows a lot of future tasks to proof all requirements listed.

## 6. References

Ackermann, J., Brinkop, F., Conrad, S., Fettke, P., Frick, A., Glistau, E., Jaekel, H., Kotlar, O., Loos, P., Mrech, H., Ortner,E., Raape, U., Overhage, S., Sahm, S., Schmietendorf, A., Teschke, T. and Turowski, K. (2002), "*Vereinheitlichte Spezifikation von Fachkomponenten",* Gesselschaft für Informatik, wi2.wiwi.uni-augsburg.de/downloads/gi-files/MEMO/Memorandum-final-2-44-mit-literatur-Web.pdf, (Accessed 18 April 2007), Page 1

Apperly, H., Hofman, R., Latchem, S., Maybank, B., McGibbon, B., Piper, D. and Simons, D. (2003), *"Service- and Component-based Development: Using Select Perspective™ and UML"*, Addison Wesley, Page 5-8

Bettin, J. (2004), *"Model-Driven Software Development, An emerging paradigm for Industrialized Software Asset Development",* www.softmetaware.com/mdsd-and-isad.pdf, Softmetaware Page 9, (Accessed 18 April 2007)

Björkander, M. (2003), *"Model-Driven Development and UML 2.0. The End of Programming as We Know It?"*, Upgrade - The European Journal for the Informatics Professional - "Software Engineering State of an Art", Page 10-13

Brooks, F. P. Jr., (1987) "No Silver Bullet: Essence and Accidents of Software Engineering", Computer 20, Chapter 10-9

Cechich, A. and Piattini-Velthuis, M. (2003), "*Component-Based Software Engineering*", Upgrade - The European Journal for the Informatics Professional - "Software Engineering State of an Art", Page 15-19

Council of European Professional Informatics Societies (2003), *"Software Engineering State of an Art"*, Upgrade – The European Journal of the Informatic Professional

Council of European Professional Informatics Societies (2004), "*Software Process Technology*", Upgrade – The European Journal of the Informatic Professional

Derniame, J. and Oquendo, F. (2004), "*Key Issues and New Challenges in Software Process Technology*", Upgrade - The European Journal for the Informatics Professional - "Software Process Technology", Page 15

Fettke, P., Intorsureanu, I. and Loos, P. (2002), *"Component oriented procedure pale in comparison "*, isym.bwl.uni-mainz.de/publikationen/wkba02_vorgehensmodelle.pdf, University of Chemnitz, (Accessed 18 April 2007)

Fowler, M. (2005), "*The New Methodology*", www.martinfowler.com/articles/newMethodology.html#SeparationOfDesignAndConstruction, (Accessed 18 April 2007)

Heckmann, B. (2007), "*Service provision in a Utility Computing environment*", 3rd International NRG Research Symposium, Network Research Group, University of Plymouth

Highsmith, J. and Cockburn, A. (2001), *"Agile Software Development: the Business of Innovation"*, IEEE ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=947100, Volume: 34, Issue: 9, Page 120 , (Accessed 18 April 2007)

ISO/IEC (2001), *"Software engineering — Product quality —Part 1:Quality mode"*, ISO/IEC, Reference number: ISO/IEC 9126-1:2001(E), Switzerland, Page 3-7

Jeffery, Chair K. and De Roure, D. (2006), *"Future for European Grids: GRIDs and Service Oriented Knowledge Utilities",* European Commission

McConnel, S. (2005), *"Code Complete German Version"***,** Microsoft Press Deutschland, Page 1-5

Meimberg, O. and Petrasch, R. (2006), "*Model Driven Architecture*", dpunkt, Heidelberg
Paulk, M. C. (2001), *"Extreme Programming From a CMM Perspective"*, IEEE, ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/52/20852/00965798.pdf?arnumber=965798, IEEE Software, Page 1-4, (Accessed 18 April 2007)

Poston, R. M. and Sexton, M. P. (1992), *"Evaluating and Selecting Testing Tools",* IEEE Software 9, 3, Page 33-42, (Accessed 18 April 2007)

Software Engineering Institute (SEI) | Carnegie Mellon University (2007), "*Component-Based Software Development / COTS Integration*", SEI Website www.sei.cmu.edu/str/descriptions/cbsd_body.html , Sections "Component qualification", "Component adaptation", "Assembling components into systems", "System evolution" , (Accessed 18 April 2007)

Stojanović, Z. (2005), *"A Method for Component-Based and Service-Oriented Software Systems Engineering"*, Delft University of Technology, Delft, Netherlands

Szyperski, C., Gruntz, D. and Murer, S. (2002), *"Component Software – Beyond Object-Oriented Programming. Second Edition"*, Great Britain, Page 21-26

Valetto, G. and Kaiser, G. E. (1995), *"Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments",* Proceedings of 7th IEEE International Workshop on CASE. Toronto, Ontario, Canada, Los Alamitos, IEEE Computer Society Press, Page 40-48