# An extended Component Lifecycle

Christian Heller
<christian.heller@tu-ilmenau.de>

Technical University of Ilmenau
Faculty for Computer Science and Automation
Institute for Theoretical and Technical Informatics
PF 100565, Max-Planck-Ring 14, 98693 Ilmenau, Germany
http://www.tu-ilmenau.de, fon: +49-(0)3677-69-1230, fax: +49-(0)3677-69-1220

**Abstract.** Based on latest achievements in the field of *Component Based Design (CBD)* and *Component Oriented Programming (COP)*, this article shows how a component lifecycle can be extended. The well-known *Hierarchical Model View Controller (HMVC)* design pattern is considered for the identification of the two concerns *Showable* and *Loadable* Following, these two concerns are added to a component to demonstrate the lifecycle extension. The component framework *ResMedLib* is introduced to show how such communication between components happens through well-defined interfaces. A summarizing reflection on the pros and cons of *Aspects* versus *Concerns* of a component will finalize the article.

**Keywords.** Component Lifecycle, Design Pattern, HMVC, Concern, Aspect, Res Medicinae, ResMedLib

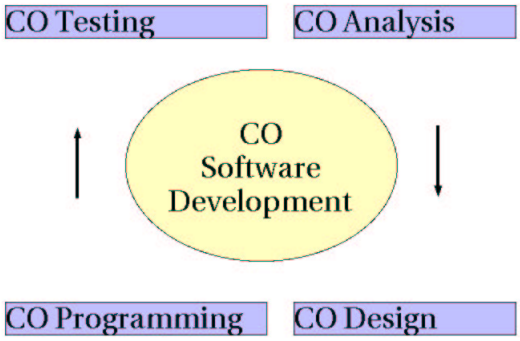


**Fig. 1.** Component Oriented Software Development.

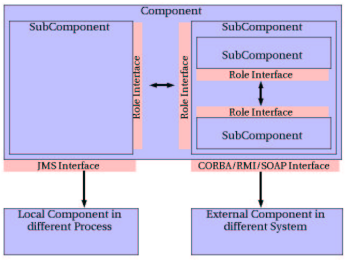


**Fig. 2.** Communication between Components.

## 1 Introduction

By following *Component Based Design (CBD)* rules, software projects try to integrate most diverse systems into one environment. Although the systems should ideally use *Component Oriented Programming (COP)* for the implementation of their components, this is not a must. Legacy systems can be encapsulated as well [Bro00], acting as one component to the outside environment. In practice, one cannot use COP without first designing with components in mind. A typical component oriented development process is displayed in Figure 1.

Each component lives in a system that is responsible for the component's creation, destruction etc. This is to be the topic of this article. What will not be considered, is the communication between remote components living in different systems (RMI, CORBA, SOAP) [blu] or processes (JMS).

When talking about components, this article sticks to the definition of [jak02] which considers components to be "a passive entity that performs a specific role". This is opposed to active components like *Agents* are. For each role, its *Interface* has to be specified to the rest of the system as shown in Figure 2.

The *Avalon* project [jak02] writes on to this:

> "... the interface is not enough. There are specific contracts that one must define and keep in mind when specifying the interfaces. In other words, what users of the component must provide, and what the component produces. When the interface and contract are defined, one can start to work on the implementation."

As can be seen, there are other concerns besides the component's role (Figure 3). In most cases, these concerns belong to a contract that is defined by the surrounding Framework.

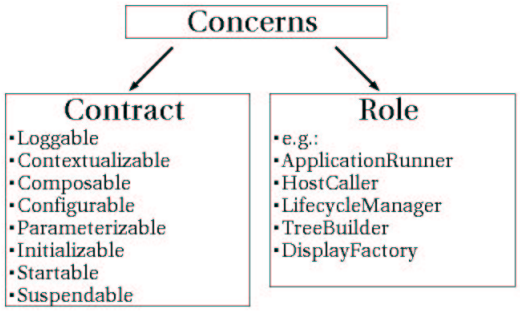


**Fig. 3.** Role and Contract Concerns.

In some way, *Concerns* are quite similar to *Aspects*. In fact, aspects also provide various outside views (interfaces) to a component (module). The *AspectJ* project [asp02] writes:

> "Consider what *Aspect Oriented Programming (AOP)* really does: It makes the modules in a program correspond to modules in the design. In any given design, some modules are optional, and some are not."

They distinguish between *Development* and *Production* aspects (Figure 4). Production aspects would be delivered with the finished product, while development aspects would be used during the development process. Often, production aspects are also used during development.

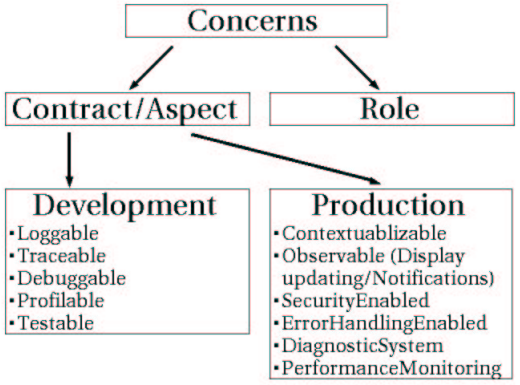


**Fig. 4.** Possible Systematics of Concerns and Aspects.

Common development aspects are used for logging, tracing, debugging, profiling or testing. Common production aspects are used for performance monitoring and diagnostic systems, for display updating or notifications in general, for security, context passing and error handling.

## 2 Basics

### 2.1 A Component Lifecycle

As latest research shows, it is highly desirable for nearly all components, even for simple objects, to implement interfaces (concerns) (Figure 5) whose methods can be called according to a given order (contract). Each interface represents a narrow view of the component or object being controlled. The order of method calls is what is known as *Component Lifecycle* (Figure 6).

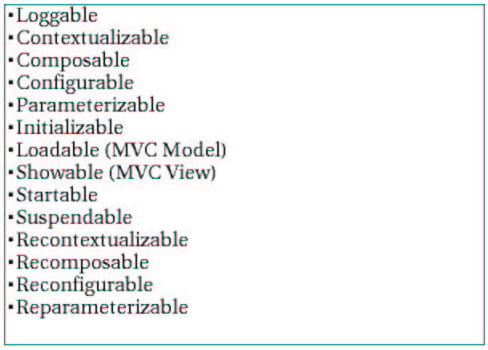


**Fig. 5.** Lifecycle Interfaces/Concerns [jak02].

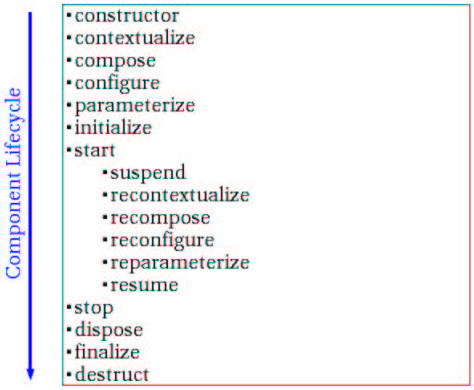


**Fig. 6.** A Component Lifecycle [jak02].

An outside, active entity is responsible for calling the component lifecycle methods in the right order. Such an

entity or *Component Container* implements the *Composable* interface and such controls and uses components. Avalon [jak02]: ”It is up to each container to indicate which lifecycle methods it will honor. This should be clearly documented together with the description of the container.”

### 2.2 The HMVC Design Pattern

It is - or, at least, should be - a common standard to use a *Model-View-Controller (MVC)* separation for the *Application* (sometimes called *Presentation*) *Layer* of a system. The advantages are at hand: A clear encapsulation of code makes it modular and easily exchangeable. Unnecessary dependencies are avoided and the system layers get decoupled.

Such flexibility allows for the introduction of changeable system parts, for the *View* as well as for the *Model*. The view, for example, might be switchable between a web frontend based on *Java Server Pages (JSP)* and a standalone *Graphical User Interface (GUI)* based on *Swing*. The data sources of a model, on the other hand, might be switchable between File, DB, CORBA/SOAP/RMI, which together comes close to a complete persistence layer.

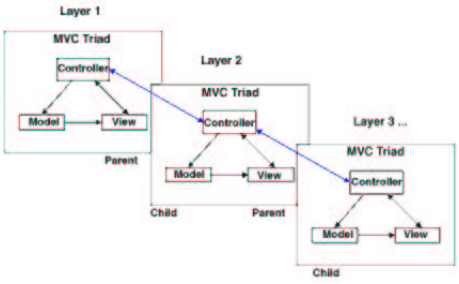


**Fig. 7.** MVC Layers [JC00].

The MVC [ea02] was extended by [JC00] to the *Hierarchical MVC (HMVC)* pattern [Figure 7]. The new idea of HMVC is to have controllers coordinating the process flow while organizing the communication between view and model. A second core idea is to allow controllers to have children. In such a manner, the hierarchy of controllers (*MVC Triads*) represents the backbone of a system whose startable root controller is called an *Application*.

Child controllers (Figure 8) should be introduced for *Frames*/*Dialogs*, *Panels* or smaller GUI components, when the controlling code gets too large. For example, a *Tree* or *Table* GUI component may become quite complex and require an own controller. However, it is not a must to create many child controllers. GUI components/panels/dialogs etc. which have no very own controller, will simply be controlled by the next higher parent controller.

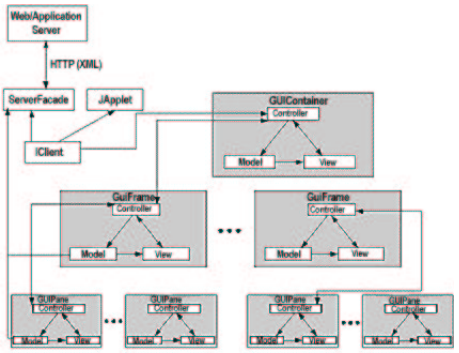


**Fig. 8.** HMVC Client Tier Instance Diagram [JC00].

## 3 An Extended Component Lifecycle

Not all components will always need to have a Graphical User Interface (GUI), as a command line, for example, is often enough to control the component. Similarly, not all components are based on a specific data model (persistence) as some components just process direct input without any data storage.

On this point, the question arises, on how to design an application component such that it has an option, but is not forced to use the complete MVC architecture?

The answer is quite obvious: Let us consider *View* and *Model* to be concerns of the component! The concerns may be called *Showable* and *Loadable*.

As described before, a component runs through several states of its lifecycle which is controlled by an outside object, called the *Component Container*. Depending on the state of several environment variables at runtime, the container may or may not call certain lifecycle methods of the component. A GUI-capable component, for example, may have the following lifecycle:

```
configure(c);
initialize();
show(v);
hide(v);
finalize();
deconfigure(c);
```

**Fig. 9.** Lifecycle of a GUI capable Component.

The source code of [Figure 9] would belong to the component container. Comparing with the previous example in [Figure 6], it can be seen that in [Figure 9], an additional method *show* has been inserted.

Just like the *configure* method gets a configuration object *c* as parameter, the *show* method gets a view object *v*

handed over, which is to be displayed. Such parameter objects could be created by the container:

```
Component c = new ComponentImpl();
View v = new ViewImpl();
c.show(v);
```

However, most often it is more suitable to let the component create these parameter objects as they logically belong together:

```
Component c = new ComponentImpl();
View v = c.createView();
c.show(v);
```

This solution has yet another advantage: A view object can be treated as component itself which means lifecycle methods will have to be called, e.g.:

```
View v = new ViewImpl();
v.configure();
v.initialize();
```

All this is much better encapsulated by a *createView* method in the GUI-capable component itself, than having the container struggling with it. If the component shall be based on a specific data model, the corresponding methods of the *Loadable* concern have to be added to the lifecycle as well:

```
Component c = new ComponentImpl();
Model m = c.createModel();
View v = c.createView();
c.load(m);
c.show(v);
```

The example becomes complete by inserting the code that checks the component for available conerns:

```
Component c = new ComponentImpl();
if (c != null) {
    if (c instanceof Loadable) {
        Model m = c.createModel();
        c.load(m);
    }
    if (c instanceof Showable) {
        View v = c.createView();
        c.show(v);
    }
} else {
    throw new NullPointerException("");
}
```

Besides the fact that these lifecycle methods can be called or not, there's another advantage: The container can determine any suitable parameter to hand over to the component. For example, there may be another view that could be displayed by the component's *show* method or another data model that might get loaded by *load*.

# 4 The ResMedLib Framework

To verify the proposed design, it has been applied and implemented within the *Res Medicinae* project [res04]. That is based on the *ResMedLib Framework* which aims to provide a modular and highly flexible structure to easily implement new components.

Two GUI-capable components have been implemented so far: The *ResMedicinae* application offering a main window to host sub components (modules) and the *Record* application meant to become a fully functional *Electronic Health Record (EHR)*. Both modules use a lifecycle similar to the example given above (Figure 9).

The *show* method is implemented by a parent class *AbstractComponent* [Figure 10] of the component and can be overridden, of course. It delegates the task of displaying the view to a *DisplayManager* which can use Window, Dialog, Frame, InternalFrame or TabPage as possible display. These displays are switchable at runtime. All this is done by factories, the details of which one can study in the code.

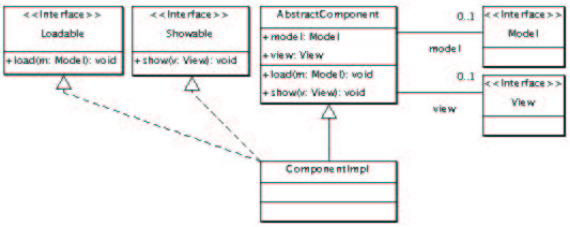


**Fig. 10.** ResMedLib class AbstractComponent in UML Class Diagram.

Due to the consistent use of the HMVC design pattern, all application controllers can be cascaded (Figure 11).

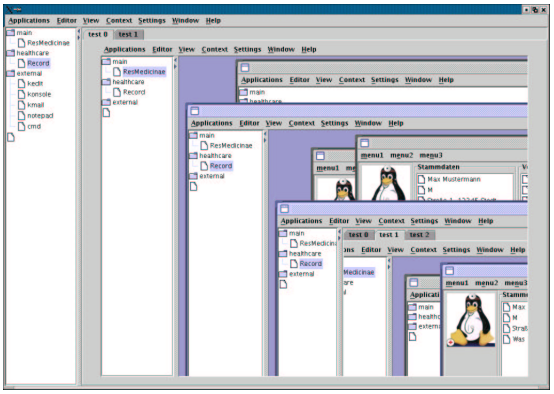

**Fig. 11.** Res Medicinae containing cascaded Sub Components.

The *Res Medicinae* project is *Free Software/ Open Source Software (OSS)*, licensed under the *GNU GPL*, i.e. the source code is freely available and extensible, as well as all documentation and other resources are - as long as derived works are free as well. Contributions and supporters are always welcome to the project :-)

## 5 Summary and Future

This article proposes the split-up of view and model of the well-known MVC design pattern into single concerns. These two, resulting concerns were integrated into a common component lifecycle of two modules in the *Res Medicinae* project. They have proven to work well and to be highly flexible due to their clear separation.

Using this advanced architecture, developers are now able to create general components which offer but do not force the use of a view (GUI) and a model (persistent data storage). This modular structure allows components to be used in most diverse environments by simply altering some of their lifecycle calls.

It seems that the use of an aspect-oriented language extension such as [asp02] can be avoided by sticking to well-defined concerns/interfaces of a component, just handing over necessary parameters. However, there's still a lot to figure out in this area. Other useful concerns will have to be identified. Existing systems are recommended to be refactored and slowly move to CBD/COP with lifecycles, to achieve higher modularity and flexibility.

## 6 Acknowledgements

My special thanks go to all these brave Enthusiasts of the Open Source Community who have provided me with a great amount of knowledge through a comprising code base to build on. I especially would like to mention the contributors of *Res Medicinae* [res04], the developers of *Scope* [sco02], of *Apache-Jakarta-Avalon* [jak02] and all the other OSS projects.

## 7 About the Author

Christian Heller has studied Electrotechnics/Biomedical Informatics at the Technical University of Ilmenau. He has worked at several small to large-sized companies, including OWiS (OTW UML tool), Intershop (e-commerce) and a big German insurance company. Besides, he is the founder of the Java-based ResMedicinae project - aiming to implement a Medical Information System - and active developer of the Open Source Community. In 2001, he returned to his former University where he plans to earn a doctorate.

## References

[asp02] *AspectJ Aspect-Oriented Java Extension*. http://aspectj.org, 2002.

[blu] *Java Blueprints*.

[Bro00] BROWN, ALAN W.: *Large-Scale, Component-Based Development*. Object and Component Technology Series. Prentice Hall PTR, London, Sydney, 2000. http://www.phptr.com.

[ea02] AL., MARTIN FOWLER ET: *Patterns of Enterprise Application Architecture (Information Systems Architecture)*. Addison-Wesley, Boston, Muenchen, 2001-2002. http://www.aw.com.

[jak02] *Apache Jakarta Avalon Framework, Web Server and Applications*. http://jakarta.apache.org, 2002.

[JC00] JASON CAI, RANJIT KAPILA, GAURAV PAL: *HMVC: The layered pattern for developing strong client tiers.* July 2000.

[res04] *Res Medicinae – Medical Information System*. http://www.resmedicinae.org, 1999-2004.

[sco02] *Scope HMVC Java Framework*. http://scope.sourceforge.net/, 2001-2002.