A flexible Software Architecture for Presentation Layers demonstrated on Medical Documentation with Episodes and Inclusion of Topological Report

Christian Heller <christian.heller@tu-ilmenau.de>, Jens Bohl <info@jens-bohl.de>,
Torsten Kunze <info@torstenkunze.de>, Ilka Philippow <ilka.philippow@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

This document describes how existing design patterns can be combined to merge their advantages into one domain-independent software framework. This framework, called Cybernetics Oriented Programming (CYBOP), is characterized by flexibility and extensibility. Further, the concept of Ontology is used to structure the software architecture as well as to keep it maintainable. Its hierarchical appearance stems from a core design decision: all framework classes inherit from one super class Item which represents a Tree. A Component Lifecycle ensures the proper startup and shutdown of any systems built on top of CYBOP.
The practical proof of these new concepts was accomplished within a diploma work which consisted of designing and developing a module called Record, of the Open Source Software (OSS) project Res Medicinae. The major task of this module is to provide a user interface for creating medical documentation. New structure models such as Episodes were considered and implemented. In this context, the integration of a graphical tool for Topological Documentation was highly demanded as well. The tool allows documentation by help of anatomical images and the setting of markers for pathological findings.
Keywords. *Design Pattern, Framework, Component Lifecycle, Ontology, CYBOP, Res Medicinae, Episode Based Documentation, Topological Documentation*

1 Introduction

Quality of software is often defined by its maintainability, extensibility and flexibility. *Object Oriented Programming* (OOP) helps to achieve these goals – but it isn't possible alone by introducing another programming paradigm. So, major research objectives are to find concepts and principles to increase the reusability of software architectures and the resulting code. *Frameworks* shall prevent code duplication and development efforts. Recognizing recurring structures means finding *Design Patterns* for application on similar problems. These two concepts – frameworks and design patterns – depend on each other and provide higher flexibility for software components [Pre94].
The aim of this work was to find suitable combinations of design patterns to compose a framework that is characterized by a strict hierarchical architecture. Everything in universe is organized within a hierarchy of elements – the human body for example consists of organs, organs consist of regions, regions consist of cells and so on. This very simple idea can also be mapped on software architectures – and basically, this is what this document is about. What kind of techniques to realize such a concept of strict hierarchy does software engineering provide? The following chapters first introduce common design patterns and the lifecycle of components as templates for own ideas and then show how the resulting framework *Cybernetics Oriented Programming* (CYBOP) [cyb04] is designed.

2 Design Principles

2.1 Essential Design Patterns

Design patterns [EG95] are elements of reusable software. They can be used for solving recurrent design problems and are recommendations on how to build software in an elegant way. With the help of these patterns, software shall be more extensible, flexible and easy to maintain with respect to future enhancements. The following patterns are essential within CYBOP.

Composite This design pattern (figure 1) allows creating tree-like object structures. One object is child of another object and has exactly one parent. This pattern is often used to realize *Whole-Part* relations: one object is *Part* of another one.

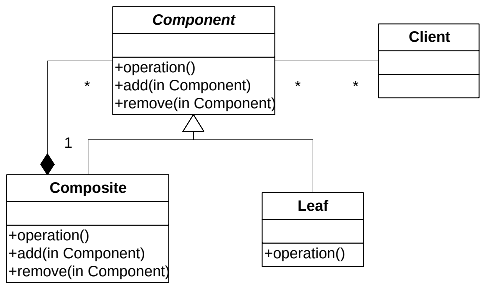

Fig. 1. Composite Pattern

Layers With the help of this pattern, software can be organized in horizontal layers (figure 2). Modules and applications can be separated into logical levels, whereby these levels should be as independent from each other as possible, to ensure a high substitutability.

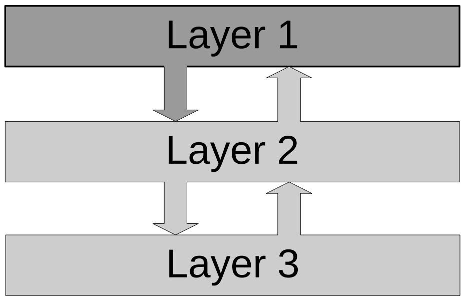

Fig. 2. Layer Pattern

Chain Of Responsibility Messages initiated by a particular object can be sent over a chain of instances to the receiving object (figure 3). So, either the message will be transmitted over a bunch of objects or evaluated immediately by the target object.

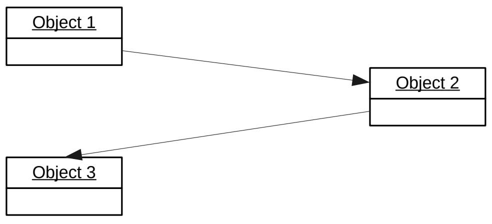

Fig. 3. Chain Of Responsibility Pattern

Model-View-Controller Dividing the presentation layers into the logical components *Model*, *View* and *Controller*, is a very approved way for designing software for user interfaces. The model encapsulates the data presented by the view and manipulated by the controller (figure 4).

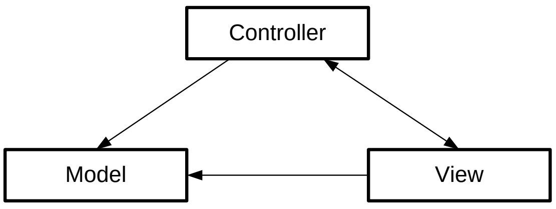

Fig. 4. Model-View-Controller Pattern

Hierarchical Model-View-Controller The Hierarchical Model-View-Controller [JC00] combines the essential design patterns *Composite*, *Layers* and *Chain of Responsibility* into one conceptual architecture (figure 5). This architecture divides the presentation layer into hierarchical sections containing *MVC-Triads*. The triads conventionally consist of model, view and controller parts. Triads communicate with each other by relating over their controller object.
Here is a short explanation of this concept, using a practical example: The upper-most triad could represent a dialog and the middle one a container such as a panel. In this container, a third triad – for example a button – could be held.

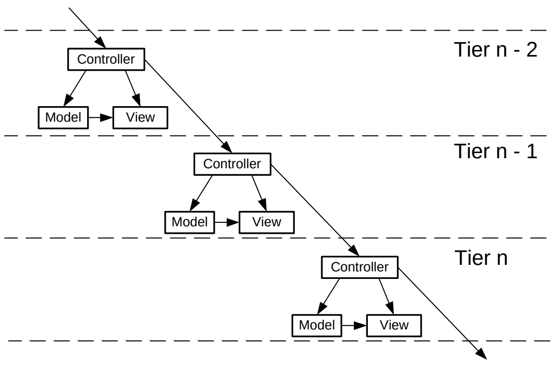

Fig. 5. Hierarchical Model-View-Controller Pattern

2.2 Component Lifecycle

Each *Component* lives in a system that is responsible for the component's creation, destruction etc. When talking about components, this article sticks to the definition of Apache-Jakarta-Avalon [jak02], which considers components to be *a passive entity that performs a specific role*.
A component has a number of methods which need to be called in a certain order. The order of method calls is what is known as *Component Lifecycle*. An outside, active entity is responsible for calling the lifecycle methods in the right order. In other words, such an entity or *Component Container* can control and use the component. The Avalon documentation [jak02] says:

> *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.*

3 An Extended Component Lifecycle

The CYBOP lifecycle of components is an extension of the lifecycle idea of Apache – basically the same idea but another background and realization.
All *Whole-Part associations* between objects were organized under the rules of the component lifecycle. The relations were created and destroyed in a sequence of lifecycle steps. These steps are realized as method calls on the components (figure 6).

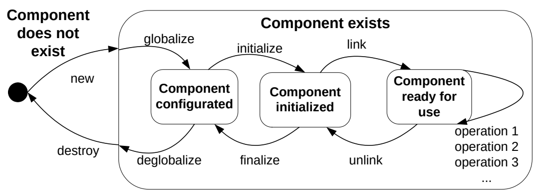

Fig. 6. State Diagram of CYBOB's Component Lifecycle

Contrary to Apache's lifecycle, this one introduces a *globalize* method by which global parameters can be forwarded throughout the whole framework. Static methods or *manager* classes such become superfluous. Analogous to the lifecycle of organic cells where the genetic information in form of a Desoxyribo Nuklein Acid (DNA) is shared before separation, the globalize method allows to forward a configuration object to any new instance.

4 Ontology

An ontology is a catalogue of types that are depending on each other in hierarchical order. It is a formal specification concerning a particular objective. CYBOP consists of three such ontologies:

– Basic Ontology
– Model Ontology
– System Ontology

Figure 7 shows the model ontology. The layer super types are `Record`, `Unit`, `Heading` and `Description`. These classes are super types of all classes in a particular ontological level.
The right side shows a concrete implementation of the model ontology – the *Electronic Health Record* [ope04]. This data structure contains all information concerning a particular patient. The figure shows `Problem` types in level `Unit`. These consist of episodes containing instances of `PartialContact`. In level `Heading`, the structural elements of a partial contact can be found – `Subjective`, `Objective`, `Assessment` and `Plan`. Therapeutical issues are placed in level `Description` – such as `Medication` with particular dose.

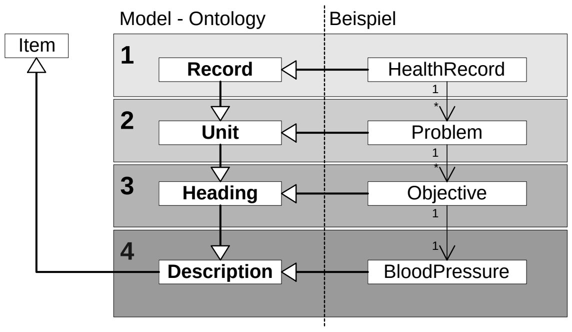

Fig. 7. Model Ontology

As shown, the concept of ontology can be used to organize data structures in a hierarchical order by defining logical layers with super types.

5 CYBOP

Section 2 introduced essential design patterns that represent the main structure of the CYBOP framework. Section 3 explained the Component Lifecycle and section 4 the well-known idea of ontology. Now these design principles and concepts will be combined to comprise their advantages and to increase the demanded quality characteristics: high flexibility and maintainability.
Structure by Hierarchy – this is the basic idea behind CYBOP. While this principle has been applied to many domain and knowledge models, especially in the field of *Artificial Intelligence* (AI), it apparently has not been used for the design of systems yet.

Let us recall the *Model View Controller* design pattern which is used in one or another form by a majority of systems, today. There is a *View* which mostly is a *Graphical User Interface* (GUI). It consists of for example a frame, panel, menubar and smaller components which are all part of the frame's hierarchy. Then, there is the *Controller*. The *Hierarchical MVC* pattern suggested to use a controller hierarchy consisting of *MVC Triads*. Finally, there is the *Model*. Not only AI systems use a *Hierarchy* to structure their domain data. The OpenEHR project [ope04] does the same.
Reflecting these facts, one question is at hand: *If View, Controller and Model ideally have a hierarchical structure, why not creating whole software systems after this paradigm?* Isn't every system essentially a *Tree* of objects?
Extending the concept of *Hierarchical Model View Controller* to whole software architectures, CYBOP was designed to be the domain-independent backbone for information systems of any kind. Originally designed for medical purposes, it should also become usable for insurance, financial or other standard applications in future.

5.1 Class Item

As shown, tree-like structures can be realized by the *Composite* pattern. In CYBOP, this pattern can be found simplified in class `Item` (figure 8) which is super type of all other classes. References, respectively relations to child elements are held within a hashmap. No attributes are used except of this hashmap. Every element of the map can be accessed by a special key value.

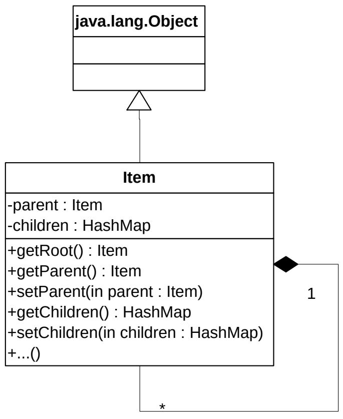

Fig. 8. Class Item

But that also means that any particular `set`- and `get`-methods become superfluous. The recommendation to encapsulate attributes produces thousands of lines of code whose usefulness is at least questionnable. In probably 90% of cases, the `set` and `get` methods consist of only one single line accessing an attribute value. Sometimes, additional lines with an update function for other parts of the system are added. They are called whenever an attribute value is changed by a `set` method:

```
public void setValue(Type v) {

    this.value = v;
    getUpdateManager().update(this);
}
```

But this update notification could as well be taken over by the parent object that was calling the `set` method on one of its child objects:

```
public void method() {

    child.setValue(v);
    getUpdateManager().update(child);
}
```

In the end, the responsibility of encapsulation falls to the super class *Item* with its access methods, alone. It is the only remaining container class in the whole framework. No other container classes have to be written ever. For the issue of sorting children (such as to simulate a *List*), other concepts have to be used which are partly unclear yet and will not be described further in this paper.
Another advantage of having just one container class is that the unpredictable behaviour in object oriented languages, when inheriting a container (hashtable) can be avoided. Find more details in [Nor].
Finally, there is the issue of security. If a system's security manager is forwarded in the *globalize* lifecycle method from object to object, as described in section 3, then it can be stored as one child of the *Item* super class. Whenever a child needs to be accessed, a parent's security manager can check for permittance.

5.2 Basic Structure

Comprising the design patterns *Composite*, *Layers*, and *Chain of Responsibility*, the CYBOP framework is comparable to a big tree containing objects organized in different levels. These levels are determined by a special *System Ontology* (as opposed to a *Knowledge Ontology* like for example OpenEHR [ope04]) and might become the topic of a follow-up paper. Figure 9 shows the object tree and the different levels of granularity.

6 Record – An EHR Module

The practical background for the application of CYBOP is *Res Medicinae* [res04]. A modern clinical information

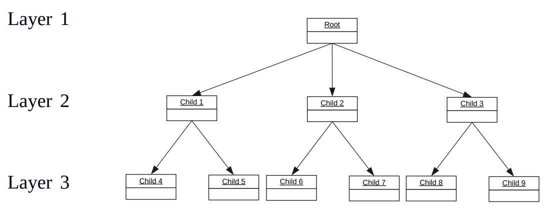

Fig. 9. Basic Structure

system is the aim of all efforts in this project. In future, it shall serve medical documentation, archiving, laboratory work etc. *Res Medicinae* is separated into single modules depending on different tasks.
One of these modules is *Record* – an application for documenting medical information (figure 10). In addition to new documentation models, it also contains a tool for topological documentation.

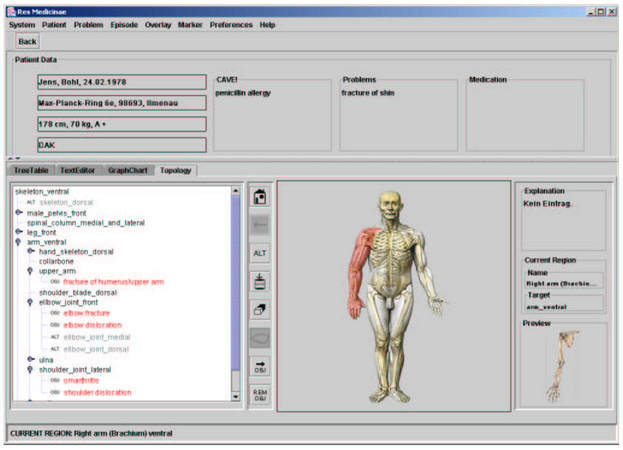
Res Medicinae
System Patient Problem Episode Overlay Marker Preferences Help
Back
Patient Data
Jens, Bohl, 24.02.1978
Max-Planck-Ring 6e, 98693, Ilmenau
178 cm, 70 kg, A +
DAK
CAVE!
penicillin allergy
Problems
fracture of shin
Medication
TreeTable TextEditor GraphChart Topology
skeleton_ventral
ALT skeleton_dorsal
male_pelvis_front
spinal_column_medial_and_lateral
leg_front
arm_ventral
hand_skeleton_dorsal
collarbone
upper_arm
OBJ fracture of humerus/upper arm
shoulder_blade_dorsal
ellbow_joint_front
OBJ elbow fracture
OBJ elbow dislocation
ALT ellbow_joint_medial
ALT ellbow_joint_dorsal
ulna
shoulder_joint_lateral
OBJ omarthritis
OBJ shoulder dislocation
Explanation
Kein Eintrag.
Current Region
Name
Right arm (Brachiu...
Target
arm_ventral
Preview
CURRENT REGION: Right arm (Brachium) ventral

Fig. 10. Screenshot of Record [urb02]

Starting from an overall view of the human body, it is possible to reach every organ or region of the body in detail (figure 11).

7 Summary

Software patterns are essential elements of frameworks. This paper showed how they can be combined to comprise their advantages and to realize hierarchical structures with unidirectional dependencies.
These structures can be created and properly destroyed using the lifecycle of components. In that lifecycle, object relations become more transparent and are easier to control and to maintain. As an extension, this paper introduced the *globalize* method by which references to globally needed objects can be forwarded through a system which eliminates the need for static methods or objects.
Ontologies can help to model particular domains and to layer software. Every level of these ontologies has a particular supertype, whereby these types depend on each other by inheritance. This concept supports the modelling and logical separation of software into hierarchical architectures. The granularity of the ontology (number of ontological levels) can be adapted to particular needs.
In the course of this document, it turned out that hierarchies are not only ideal for structuring domain knowledge but also for structuring whole systems (applications). As essential design decision to take, this paper proposed to introduce a top-most super class (called *Item* here) which represents a *Map* container. Thousands of lines of code in form of *set* and *get* methods can such be eliminated which leads to a tremendous code reduction and improved clarity.
By applying the new concepts introduced in this document, the quality of software can hopefully be increased. The time for building systems might get reduced. The clear architecture should avoid common confusion as the systems grow.

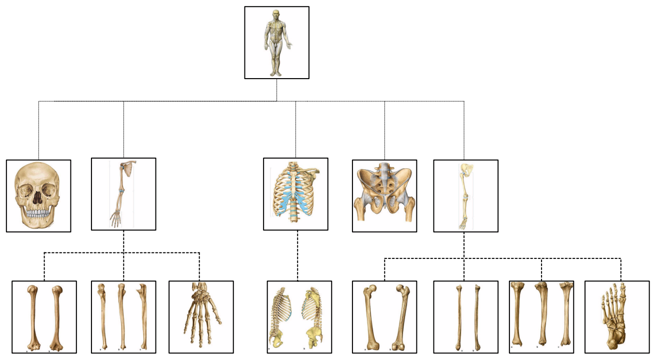

Fig. 11. Excerpt from Topological Structure of Human Skeleton

8 Acknowledgements

Our special thanks go to all Enthusiasts of the Open Source Community who have provided us with a great amount of knowledge through a comprising code base to build on. We'd also like to acknowledge the contributors of *Res Medicinae*, especially all medical doctors who supported us with their analysis work [CH04] and specialised knowledge in our project mailing lists. Further on, great thanks goes to the Urban and Fischer publishing company, for providing anatomical images from their *Sobotta – Atlas der Anatomie*.

References

[CH04] Christian Heller, Karsten Hilbert, Roland Colberg et al.: *Analysedokument zur Erstellung eines Informationssystems fuer den Einsatz in der Medizin*. http://www.resmedicinae.org/model/analysis, 2001-2004.

[cyb04] *CYBOP – Cybernetics Oriented Programming*. http://www.cybop.net, 2002-2004.

[EG95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (Gang Of Four): *Design Patterns. Elements of reusable object oriented Software.* Addison-Wesley, Bonn, Boston, Muenchen, 1 , 1995. 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.* Java World, July 2000. http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc$_p.html$.

[Nor] Norvig, Peter: *The Java IAQ: Infrequently Answered Questions*. http://www.norvig.com/java-iaq.html.

[ope04] *OpenEHR – Design Principles Document*. http://www.openehr.org, 2001-2004.

[Pre94] Pree, W.: *Meta Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design*. *Proceedings of ECOOP '94*, 150–162, 1994.

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

[urb02] *Anatomical Images from Sobotta: Atlas der Anatomie*, 2002. http://www.urbanfischer.de.