A flexible Software Architecture for Communication

Christian Heller, Torsten Kunze, Jens Bohl, Ilka Philippow
<christian.heller@tu-ilmenau.de>, <info@TorstenKunze.de>,
<info@jens-bohl.de>, <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 extended and 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. A* Component Lifecycle *ensures the proper startup and shutdown of any systems built on top of CYBOP.*
One core component of the framework is Layer PerCom *- an advanced architecture for seemless integration of communication paradigms and persistence mechanisms. Overcoming the classical scheme of thinking in terms of* Domain, Frontend, Backend *and* Communication*, PerCom treats them all similar, as passive data models which can be translated into each other - as opposed to the classical approach that unnecessarily complicates their design.*
The practical proof of this combined architectural approach was accomplished within the diploma work of Torsten Kunze [Kun03] which consisted of an (ongoing) effort to design and develop a module called ReForm*, for the* Open Source Software *(OSS) project* Res Medicinae*. The major task for this module is to provide a user interface for printing medical forms. It was used to examine the communication between modules and to find a structure for effective implementation and easy expansion.*
Keywords. *Design Pattern, Framework, Persistence, Communication, User Interface, Frontend, Backend, CYBOP, Res Medicinae*

1 Introduction

Quality of software is often defined by its maintainability, extensibility and flexibility. *Object Oriented Programming* (OOP) was to help to achieve these goals – but this wasn't possible by only introducing another programming paradigm.
So, major research objectives are to find concepts and principles to increase the reusability of software architectures and of 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].
This paper is not about frameworks but our solutions are extracted and used in one called *Cybernetics Oriented Programming* (CYBOP) [Hel02a]. Its main concept is based on the hierarchical structure of the universe. This very simple idea can perfectly be mapped on software systems. The aim of this work was to find an architectural approach that simplifies and unifies the implementation of any kind of communication mechanism, may it be for persistence, communication with remote systems or user interaction.
The first chapters elucidate three common design patterns that were used as a basis to build on. The same sections describe all suggested modifications to these patterns. Following chapters show details of the new design and how to integrate them into physical architectures. Practical proof was given in form of the *ReForm* module described before the final summary.

2 Design Principles

2.1 Data Mapper Pattern

Originally, the *Layer PerCom* approach of CYBOP was based on the *Data Mapper* pattern (figure 1).

It is part of Martin Fowler's pattern collection called *Enterprise Application Architecture* [Fow02]. The most important idea of this pattern is to abolish the interdependency of domain model and database (persistence).
The arrows in figure 1 indicate the direction of dependency. Each domain model class knows its appropriate persistence finder interface but does not know their implementation, i.e. how data are actually retrieved from the database. The data mapper implementation is part of the mapping package that implements all finders and maps

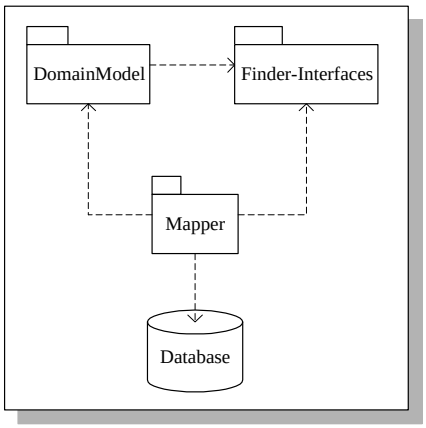

Fig. 1. The Data Mapper - Pattern [Fow02]

all data of the received result set to the special attributes of the domain model objects.
There is no need for the domain model to know where the database is located or how to get the data – and also not how to map the entity-relationship model data. If all these things are done by the corresponding data mappers, why shouldn't it be possible to get such a mapping package for persistence media of any kind, no matter which communication paradigm (File Stream, JDBC, ODBC with SQL) is used? Users would have a number of persistence mechanisms to dynamically choose from; Developers would not have to implement the same mechanisms again and again for each new Module (Application) – leading to clearer code with greatly reduced size.
This functional code separation would make it very easy to develop a complicated domain model and update it later, if necessary. The data mapper package could contain special parts for local storage in a file system, in various file formats such as XML, CSV, RTF, TXT etc. (whether it makes sense or not to store domain data in a pure text file), for a number of databases (relational, object relational, object oriented) and so on. Each of those specialised parts knows how to communicate with its appropriate persistence medium and only with it. They all include a specialised mapper class, called *Translator* in CYBOP. It translates the data from the domain model to the model of the corresponding persistence mechanism.

2.2 Data Transfer Object Pattern

It is a well-known fact that many small requests between two processes, and even more between two hosts in a network need a lot of time. The local machine with two processes has to permanently exchange the program context and the network has a lot of transfers. For each request, there is at least a necessity of two transfers – the question of the client and the answer of the server.
Transfer-methods are often expected to deliver common data such as a Person's address, i.e. surname, first name, street, zip-code, town etc. These information is best retrieved by only one transfer-call. That way, the client has to wait only once for a server response and the server does not get too many single tasks. In the address-example, all address data would best be packaged together and sent back to the client.

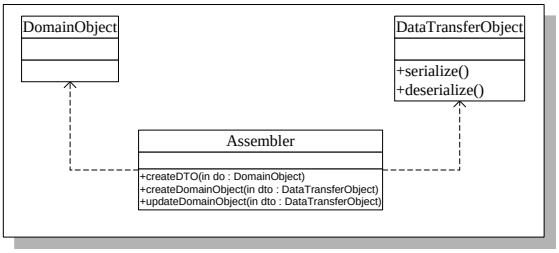

Fig. 2. The Data Transfer Object - Pattern [Fow02]

And that is exactly what the Data Transfer Object pattern (figure 2) proposes a solution for: A central *Assembler* class takes all common data of the server's domain model object and assembles them together into a special object called *Data Transfer Object* (DTO) which is a flat data structure. The server will then send this DTO over network to the client. On the client's side, a similar assembler takes the DTO, finds out all received data and maps (disassembles) them to the client's domain model. In that manner, a DTO is able to drastically improve the performance in communications.
Comparing with the Data Mapper from chapter 2.1, the assembler's task of translating between data models seems quite similar if not the same. Hence, why shouldn't it be possible for inter-system-communications over network to use a *Translator* similar to the one for persistence? This translator could provide special parts for assembling different types of DTOs, independent from which communication protocol/language (Sockets, RMI, JMS, CORBA, SOAP etc.) is used.

2.3 The Model View Controller Pattern

After having had a closer look at common design patterns for persistence and communication, this section finally considers the so called *Frontend* of an application which is mostly realized in form of a graphical user interface.
Nowadays, the well-known *Model View Controller* design pattern (figure 3) is used by nearly all standard applications. Its principle is to have the *Model* holding domain data, the *View* accessing and displaying these data and the

Controller providing the workflow of the application by handling any signals (actions) appearing on the view.

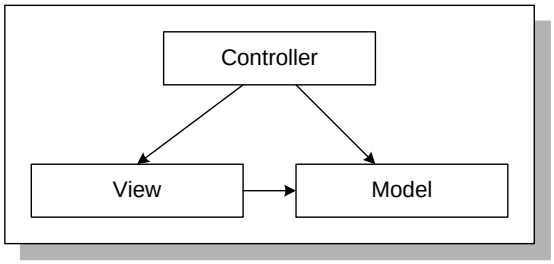

Fig. 3. Classical MVC

Since the view (graphical user interface) serves as means of communication between a software system (application) and its user (Human Being as system), the view is in fact just another type of communication model that should be assembled by a special translator.
Because there are many ways in which domain data can be displayed, different user interfaces can exist. Each of them has to have its very own translator item that knows how to map data both ways, from the domain model to the user interface model and vice-versa.

3 Layer PerCom

The following considerations are based on the design pattern extensions introduced in section 2. The terms *Mapper* and *Assembler* are converted and merged into the term *Translator*.

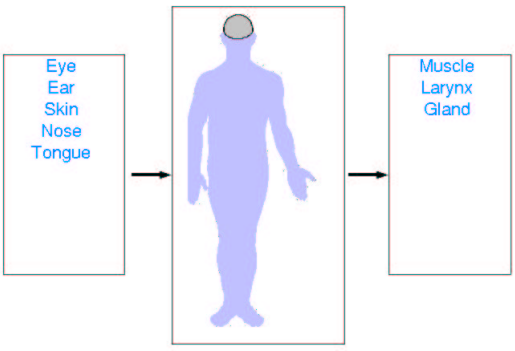

Fig. 4. Human Being as System (Body) with Model (Brain)

Following the CYBOP approach, nature - in our case the Human body - will be considered next. Humans have organs responsible for information input and output (figure 4). In between input and output, the information is processed by the brain that contains a specific abstract model of the surrounding real world. The human brain consists of several regions, each being responsible for a special task, such as the optical region for seeing or the cerebral cortex for actual information processing which possibly leads to awareness.

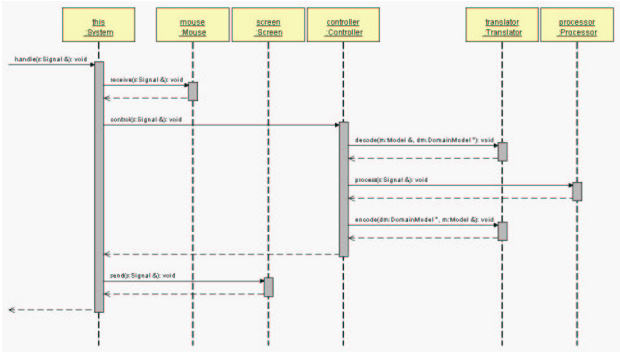

Fig. 5. Signal Processing as UML Sequence Diagram

The following example is to demonstrate a typical information processing procedure (figure 5): One human *System* wants to send another human *System* a message. It decides for an acoustical *Signal*, formulates a sentence and talks to the other human *System* (*handle* method). The other human receives the *Signal* across its ear organ (*Keyboard*, *Mouse*, *Network*). The *Signal* is then forwarded to the receiver's brain (*Controller*) where a special *Region* responsible for acoustics (*Translator*) translates the data (*decode* method) contained in the *Signal* and sorts them into the human's abstract model of the surrounding real world (*DomainModel*). Processing of the signal happens in the cerebral cortex of the brain (*Processor*). If the addressed listener wants to send an answer *Signal*, it may do so by triggering a muscle reaction. For this to happen, the motoric brain region (*Translator*) needs to translate (*encode* method) abstract model (*DomainModel*) data into a special communication model for the answer signal. Finally, the answer signal will be sent as muscle action (data display on *Screen*).
As can be seen, there is always a translator that is able to map domain model data to communication model data (*encode* method) and back (*decode* method). Depending on which communication medium is used, different translators need to be applied (figure 6).

Every system has exactly one domain model but communication models of arbitrary type can be added anytime

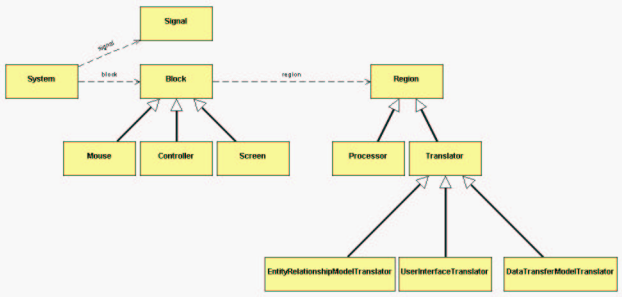

Fig. 6. Translator Classes in a UML Class Diagram

(figure 7). Every translator knows only how to translate between the domain model and a special communication model. Direct translation between communication models is forbidden as it would break the flexibility of the whole framework. In other words, translations always have to be done via the domain model.

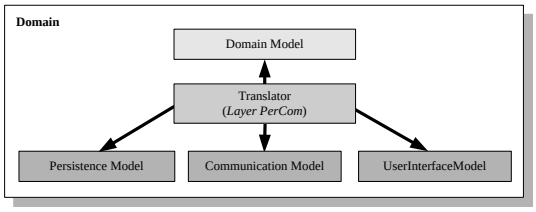

Fig. 7. Translators accessing various Models

This highly flexible and extensible architecture is absolutely transparent to the user. S/He will not know whether the current communication is with the local file system, a database or a remote process on another machine. However, this transparency causes a number of problems. Surely, the most common question is how to ensure consistence, security and minimum redundance? The following two sections give an answer to the first part of this question – consistence and uniqueness of data sets. Maximizing security and minimizing redundancy have to be analysed in future works.

3.1 Object ID (OID)

Most database systems provide an own algorithm to generate primary keys for the tables. But the applications that use the *Layer PerCom* architecture shall also be able to work if a database server is not reachable, e.g. due to a network failure. Thats why the keys are generated locally, by each application. Based on the assumption that every host in a network has a network card, it thereby has a unique internet address. This number is concatinated with an exact time stamp (nanoseconds). That is why the OID is unique in the global network and unique in time.
The proposed approach uses the OID as file name for local storage and the same OID as primary key in the main table of the database. Therewith, both models can be mapped to each other. Of course, it is necessary to avoid overwriting of new data in the database. If, for example, a network connection is cut and a little later, one wants to get data from the local files and write them up in the restored central database, it has to be made sure that nobody else has modified the data during the offline-time. That is why there is another technique to ensure this – the time stamp.

3.2 Time stamp

Most database developers will know this technique. Each table has a separate column for storing the time at which the data were written into this table. If someone requests information from the database, the time stamp is delivered as well. After modifying the data, they have to be written back into the database. At this time, both timestamps (the one in the database table and the one delivered before) are compared. If there is a difference, the data were modified by another user. Then, one has to care about the update without overwriting the new data in the table.

4 Physical Architecture

4.1 Two Tier Model

The proposed architecture CYBOP/PerCom is currently implemented in form of a Two-Tier-Architecture (figure 8).

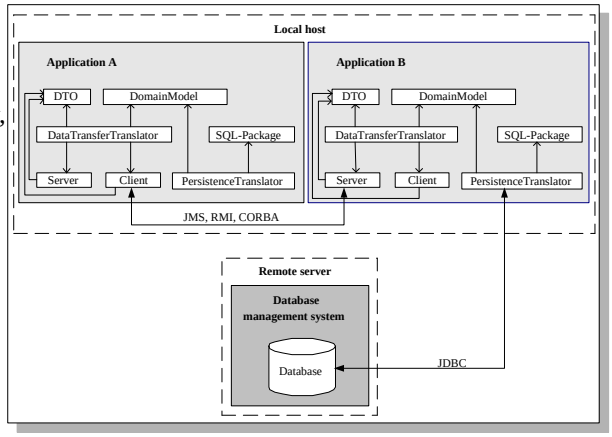

Fig. 8. Two-Tier-Architecture

It shows how two autarchic components (Application A and B) intercommunicate and save their domain data in

different ways. Each component fulfills a special task and works as a client as well as a server. One can recognize the two patterns *DataMapper* and *DTO*.
If a client requests some data from another component, the central object *DataTransferTranslator* collects all needed information from the *DomainModel* and encodes (packs) them into one *DataTransferModel*. Now, the *Server* object can send this *DTO* back to the requesting client component. On the other side of the wire, the *Client* object receives the *DTO*, a *DataTransferTranslator* decodes (unpacks) the data and writes them into the *DomainModel*.
In this example, the two components are located on the same host. It is also possible to distribute them. Therefore, each component is also able to communicate with other components that are situated somewhere in the network. The arrows in applications indicate the dependencies between the single architectural elements, whereas the outside arrows show the communication between components and database server.
All data storing operations are hidden in a special *PersistenceTranslator* like the one shown in figure 8, on the example of a database. The SQL statements were placed in a separate package. If there is the need for getting information from a database, the translator uses the statements of the *SQL-Package* and maps data of the result set to the *DomainModel*.

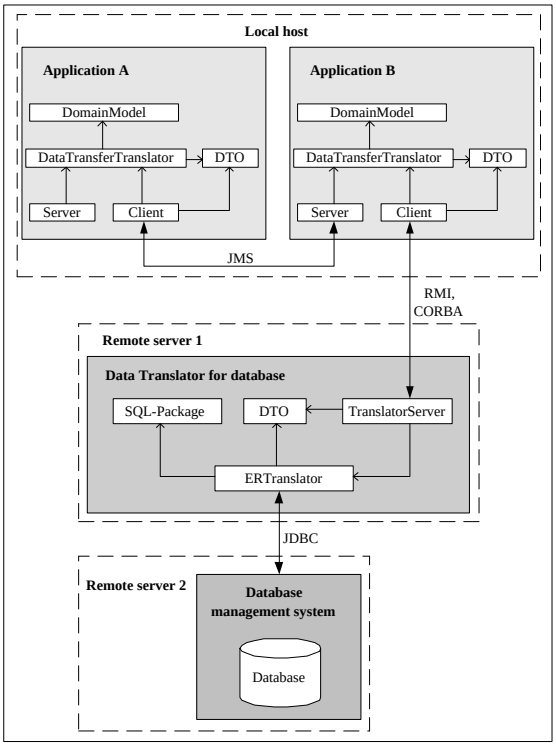

Fig. 9. Three-Tier-Architecture

4.2 Three Tier Model

In section 4.1, a typical Two-Tier-Architecture was shown. But to provide a more comfortable structure, there is the necessity of a Three- or Multi-Tier-Architecture. If, for example, the location of the database server was changed then, in a Two-Tier-Architecture, all clients would have to be updated. Figure 9 makes a proposition to solve this problem.

5 ReForm Module

The practical background for the application of CYBOP is *Res Medicinae*. A modern clinical information system is the aim of all efforts in this project. In the future, it shall serve medical documentation, laboratory data, billing etc. *Res Medicinae* is separated into single modules solving different tasks.
One of these modules is *Record* – an application for documenting medical information. In addition to new documentation models, it also contains a tool for topological documentation (figure 10).

A second module in which the *CYBOP/Layer PerCom* concepts were applied, is *ReForm* (figure 11). It offers a number of medical forms that can be filled in and printed out. Since one of the main reasons to implement this module was the testing and proof of the new persistence and communication concepts, it includes a dialog for choosing the communication protocol or persistence mechanism, respectively. This is the only remaining part where users have to care about the underlaying techniques. They also have to decide whether to use the local file system via XML-format or to store the data in a central database. In the future, the same XML file format may as well be used for remote communication, e.g. via SOAP.
Because of the component-based architecture of Res Medicinae, it is possible to start more than one instance of ReForm. In this way, the data exchange between modules can be tested. A module X looks for another registered module Y at the naming service (of RMI, CORBA or some other). X gets the address of the remote service Y (depending on the communication mechanism). The stub and skeleton of X and Y marshal, send and unmarshal the data for further working.

6 Summary

Persistence, communication and user interface mechanisms have common properties. Classical system architectures treat them as *Backend*, *DataTransfer* and *Frontend* and use different methods and design patterns (*DataMapper*, *DataTransferObject*, *ModelViewController*) to implement them.

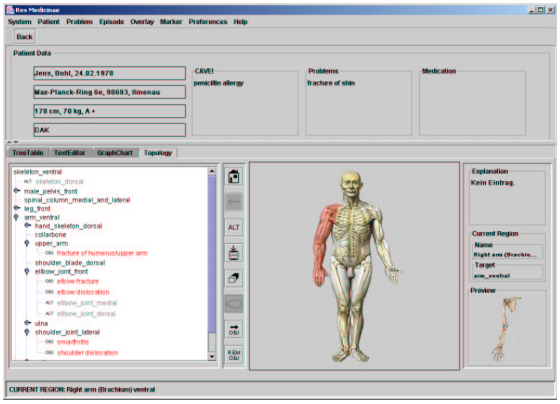

Fig. 10. Record Module [urb]

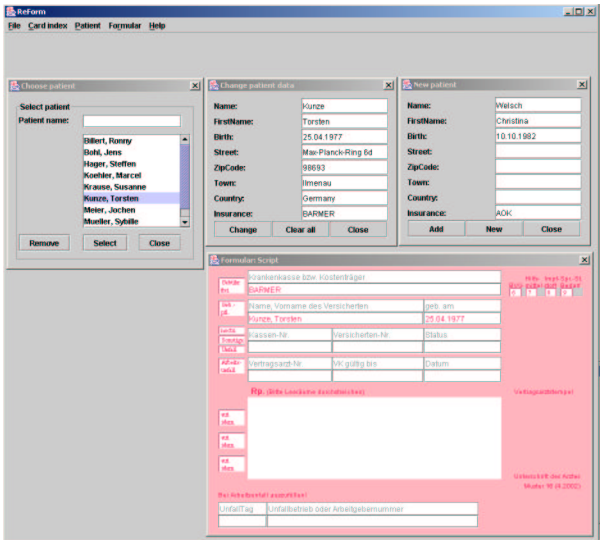

Fig. 11. ReForm Module

This paper proposes to sum up their common properties and behaviour in parent classes and to merge them all into one architecture, avoiding redundant parts. The resulting design is a good solution for the implementation of highly flexible, easily extensible and maintainable, reusable source code. The interdependency of domain data, persistence layer, communication layer and user interface is abolished. The time needed to create such an architecture (like in form of the CYBOP framework) is more than for the classical way. But once the architecture is there – it can save a tremendous amount of time when deriving modules being capable of communicating across various mechanisms. Due to its flexibility and low dependencies, it also ensures that extensions (e.g. new communication mechanisms) and modifications can be done anytime later without destroying already existing solutions.

7 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 [Hel02b] 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*.

8 About the Authors

Christian Heller, born in 1971, 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. He is founder of the Java-based *Res Medicinae* 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.

Torsten Kunze, born in 1977, has been studying computer science from October 1997 until January 2003 at the Technical University of Ilmenau with subsidiary subject *Electronic Media Engineering*. During this period, he worked six months at *Siemens AG* in Munich. He is member of the *Res Medicinae* project and active developer of the Open Source Community.

References

[CH02] Roland Colberg und Peter Hahn Christian Heller, Karsten Hilbert. *Analysedokument zur Erstellung eines Informationssystems fr den Einsatz in der Medizin*. 2002.

[Fow02] et al. Fowler, Martin. *Pattern of Enterprise Application Architecure*. Addison-Wesley, 2002.

[Hel02] Christian Heller. Cybernetics oriented programming. http://resmedicinae.sourceforge.net, 2002.

[Kun03] Torsten Kunze. Untersuchung zur realisierbarkeit einer technologieneutralen mapping-schicht fuer den datenaustausch am beispiel einer anwendung zum medizinischen formulardruck als integrativer bestandteil eines electronic health record (ehr). Master's thesis, Technische Universitt Ilmenau, Januar 2003.

[Pre94] W. Pree. Meta patterns – a means for capturing the essentials of reusable object-oriented design. In *Proceedings of ECOOP '94*, pages 150–162, 1994.

[urb] Urban und fischer verlag, muenchen. http://www.urbanfischer.de/.