An Investigation on the Applicability of Inter-Disciplinary Concepts to Software System Development

Christian Heller <christian.heller@tuxtax.de>
Ilka Philippow <ilka.philippow@tu-ilmenau.de>

Technical University of Ilmenau
Faculty for Computer Science and Automation
Institute of Technical Computer Science
PF 100565, Max-Planck-Ring 14, 98693 Ilmenau, Germany

Abstract. This article reports about an effort trying to improve software system design by applying to it concepts taken from other scientific disciplines. The resulting programming theory differs from traditional ones. It is based on firstly, a strict distinction of statics and dynamics, secondly a knowledge schema structuring models and their meta information hierarchically, and thirdly the separation of state- and logic knowledge. Many problems existing in classical programming paradigms and languages are solved in this theory.
Keywords: Software Design, Knowledge Schema, CYBOP, Cybernetics, Programming

1 Introduction

Software Development as one field of the science of *Informatics* has delivered many innovative concepts in its meanwhile rather long history. While some of them (like procedural-, but also object-oriented programming) are already in use for decades, others (like concerns, components or ontologies) are still quite young and yet have to prove their suitability for certain tasks of software development, or yet have to get accepted by the business world.

This article reports about the results of a five-year-long research work within the area of software development, in particular software design.

1.1 Software Crisis

An early question in software engineering was how to write programs that control a computer system's *Hardware* correctly and efficiently. Over time, the importance of hardware shifted in favour of *Software* which nowadays contains most of the logic needed to run an application on a computer system. Consequently, much more research emphasis is now placed on the finding of clever modelling concepts that help writing correct and effective, stable and robust, flexible and maintainable, secure software. Another objective is to increase the effectiveness and lessen the expenditure of cost and time in software development projects, by *reusing* (pieces of) software.

The past 40 years have delivered numerous helpful concepts, for instance *Structure* and *Procedure*, *Class* and *Inheritance*, *Pattern* and *Framework*, *Component* and *Concern*, and many more. They undoubtedly have moved software design far forward. Nevertheless, the dream of true componentisation and full reusability has not been reached. Czarnecki [10] identifies problems in the four areas: *Reuse*, *Adaptability* (*Flexibility*), management of *Complexity* and *Performance*.

Modern software is very *complex*. It runs on different hardware platforms, uses multiple communication paradigms and offers various user interfaces. Many tools and methods assist experts as well as engineers in creating and maintaining software but do they not seem sufficient to cope with the complexity so that often, systems still base on buggy source code causing:

- False Results
- Memory Leaks
- Endless Loops
- Weak Performance
- Security Holes

Are these exclusively the fault of software developers? Or, are the used concepts perhaps insufficient? Using the same, allegedly unsatisfying concepts caused some people to talk about an ongoing *Software Crisis*, sometimes *Complexity Crisis*, affecting not only high-level application programming, but also low-level microchip design [11].

However, answers are not easy to find. Software design is *Arts* and *Engineering*, at the same time. Not everything is or can be regulated by rules. It is true, developers have to stick to a set of design rules – and tools that support their usage exist – but they also have to be very creative. All the time, they have to have new, innovative ideas and apply them to software. This is what makes the creation, integration, test and maintenance of software so difficult. There is not really a uniform way of treating it.

1.2 Motivation

To the issues that the work described in this article had with some state-of-the-art solutions belong three things:

1. *Abstraction Gaps* in Software Engineering Process (section 2.1)
2. *Misleading Tiers* in Physical Architecture (section 2.2)
3. *Modelling Mistakes* in Logical Architecture (section 2.3)

The traversing of abstraction gaps in a software engineering process belongs to the main difficulties in software development, and causes considerable cost- and time effort. It necessitates a steady synchronisation between domain experts and application system developers, because their responsibilities cannot be clearly separated and interests often clash. A first objective was therefore to contribute to closing these gaps, especially the one existing between a designed system architecture and the implemented source code.

The misinterpretation of the physical tiers in an information technology environment often leads to wrong-designed software architectures. Logical layers are adapted to physical tiers (frontend, business logic and backend) and differing patterns are used to implement them. Instead, systems should be designed in a way that allows the usage of a unified translator architecture, so to give every application system the capability to communicate universally by default, which was the second objective.

Several well-known issues exist with the modelling of logical system architectures, for example: fragile base class problem, container inheritance, bidirectional dependencies, global data access. These and others more result from using wrong principles of knowledge abstraction, like the bundling of attributes and methods in one class, as suggested by *Object Oriented Programming* (OOP), or the equalising of structural- and meta information in a model. A third aim was therefore to closer investigate the basic principles and concepts after which current software systems are created, and to search for new concepts, with the objective of finding a universal type structure (knowledge schema).

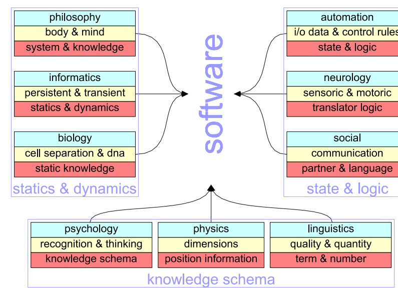

Fig. 1. Mindmap of Influential Sciences

1.3 Idea

On its search for new concepts, this work intentionally tried to cross the borders to other scientific disciplines. The idea behind is as simple as it is helpful: *Inspect solutions of various other disciplines of science, phenomenons of nature, and apply them to software engineering ... in order to find out if weaknesses existing in traditional techniques can be eliminated.* Since results from many different sciences were applied to software engineering, the work can be called an *inter-disciplinary* effort. Most emphasis, however, was placed on the comparison between human- and computer systems, which is why this work was given the name *Cybernetics Oriented Programming* (CYBOP). Nature has always been a good teacher and its principles have often been copied; so did this work.

Figure 1 shows some of the sciences whose principles were considered in this work. The name of a field of science is shown on top of each box. Made observations are mentioned below, in the middle. The resulting design recommendations for software can be found at the bottom of each box. The recommendations are grouped into those that justify a separation of *Statics and Dynamics* (left-hand side), a new kind of *Knowledge Schema* (lower part of the figure) and a distinction between *State and Logic* models (right-hand side).

It has to be mentioned though, that only some of the principles underlying a specific field of science were considered in the figure and in more detail later in this article. The figure does by no means claim to be complete. The shown observations are only those that seemed promising in the context of software design. The existence of persistent and transient data, for example, is only one of many aspects of the science of informatics. Similarly is the existence of sensoric and motoric nerve system just one aspect of the field of neurology. And so on. Further details on the mentioned sciences and observations are not given here, since later sections will elaborate on some of them.

1.4 Method

The work described in this article was undertaken in form of *Constructive Development*, as method of research. That is, an application prototype *Res Medicinae* (section 4.3) for use in the medical domain was developed in parallel to the actual theoretical investigations.

Prototype development started off by creating a state-of-the-art software architecture using *Object Oriented Programming* (OOP) principles and the *Java* programming language. When the first design problems occured, these were solved by applying suitable software patterns – mainly those of [14, 5, 13]. The steady search for a flexible architecture with only few dependencies then lead to the restructuring of the application prototype, according to the recommendations of *Component Oriented Programming* (COP) with *Concern Interfaces*, as suggested at that time by the *Apache-Jakarta-Avalon* project [2].

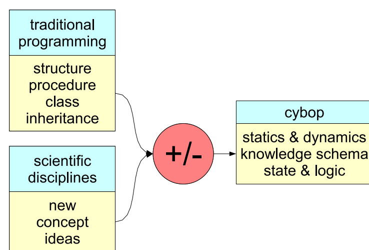

Fig. 2. Merger of Concepts

However, these refactorings were only some of at least two dozens, since also COP and the application of concerns, as well as other concepts applied later (e.g. ontological structure implemented using the means of OOP) turned out to have their deficiencies. According to the idea mentioned before, traditional concepts were thus complemented, merged or revised with new concepts stemming from other scientific disciplines (figure 2), whenever a classical design solution became unsatisfying.

Over the creation of a framework called *ResMedLib*, which encapsulated general application functionality, the prototype development finally ended up in a complete reengineering: most of the functionality formerly residing in the framework was moved into an interpreter (section 4.2) written in the *C* programming language; the actual application knowledge, on the other hand, was put into special files, for which an *Extensible Markup Language* (XML)-based language (section 4.1) was defined.

Since problems did not occur in a predictable way, while developing the mentioned prototype application, their presentation in order of appearance would be rather confusing. An adapted structure of sections is therefore used in this article, which first describes a number of observed discrepancies (section 2), then reflects on the most essential new concepts (section 3), before it later explains how these were implemented in practice (section 4).

2 Existing Problems

The problems elaborated on following belong to the software engineering process (abstraction gaps), to the physical architecture (misleading tiers) as well as to the logical architecture (modelling mistakes) of systems.

2.1 Abstraction Gaps

Software has to be developed in a creative process called *Software Engineering Process* (SEP) or *Methodology* (figure 3).

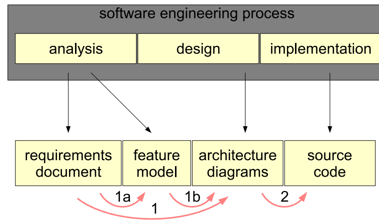

Fig. 3. Abstraction Gaps

Different forms of SEP exist: *Waterfall*, *Iterative*, *Extreme Programming* (XP) and *Agile Programming*. But every project, consciously or not, follows a SEP that sooner-or-later, in one form or the other, goes through three common phases: *Analysis*, *Design* and *Implementation*. Each phase creates its own model of what is to be abstracted in software and it is the differences in exactly these models that often cause complications.

A previous article [16] mentioned the *Requirements Document*, *Feature Model*, *Architecture Diagrams* and *Source Code* as forms of knowledge abstraction. It also described the following abstraction gaps (see figure 3) that have to be crossed:

1a Requirements Document/Feature M.
1b Feature Model/Architecture Diagr.
2 Architecture Diagrams/Source Code

By improving the *Traceability* between requirements and the architecture, feature models (known from system family/ product line engineering) contribute to minimising gap 1. Together with architecture diagrams, they ease communication between stakeholders in the SEP, because of their human-readable form and implementation-independence. But sooner-or-later, also these have to be transferred into source code, by crossing gap 2.

Bridging or closing these abstraction gaps (sometimes called *Semantic- or Conceptual Gaps*) is also known as: *achieving higher intentionality* and remains an unsolved task for software engineering. One aim of the work described in this article was to contribute to a possible solution, with focus on *reducing* gap 2, existing between a designed architecture and the implemented code.

2.2 Misleading Tiers

When distinguishing human- and technical systems, the kinds of *Communication* are:

- Human ↔ Human
- Human ↔ Computer
- Computer ↔ Computer

Each of these relies on different techniques, transport mechanisms, languages (protocols) and so on. But the general principle after which communication works, is always the same – no matter whether technical *Computer* systems or their biological prototype, the *Human Being*, are considered: Information is *received*, *stored*, *processed* and *sent*. Despite these common characteristics, today's *Information Technology* (IT) environments [18] treat communication between a computer system and a human being differently than that *among* computer systems.

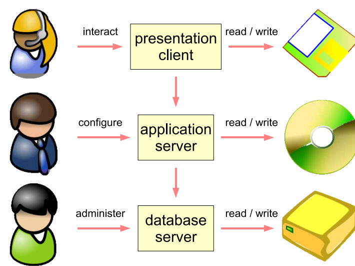

Fig. 4. Universal Communication

Figure 4 shows a three-tier environment: tier 1 represents the *Presentation Layer*; tier 2 stands for the *Application Layer*; tier 3 is the *Database (DB) Layer*. Typical synonyms are, in this order: *Frontend*, *Business Logic* and *Backend*. The tiers (layers) serve two needs: connect different locations and share work load (*Scaling*). However, the split into tiers of that kind raises two illusions:

1. *Users only interact with clients*
2. *Persistent data are stored in DB only*

Many IT architectures, or at least their illustrations, neglect the fact that in reality *all* systems need a *User Interface* (UI), for at least being administered by humans, and *almost* all systems, even *Database Management Systems* (DBMS) themselves, store some of their persistent data outside a database, for example locally available configuration information. This is not necessarily a problem for the IT environment as such, but it is for the internal architecture of software systems. Special solutions have to deal with frontend (UI framework), business logic (domain patterns) and backend (data mapping), and often additional mechanisms for local and remote communication. The serious differences in these design solutions are one root of well-known problems like multi- directional inter-dependencies between system parts, that make software difficult to develop and hard to maintain.

One aim of the work described in this article was to investigate possibilities for a *unification* of communication paradigms, that is high-level design paradigms rather than low-level protocols, in order to architect software in a way that allows the computer system it runs on to communicate *universally*.

2.3 Modelling Mistakes

Most modern software is not written directly in a machine language but designed in form of higher-level models instead. These allow to speed up application development and help avoiding errors. *Object Oriented Programming* (OOP), for example, uses design concepts like the *Class* owning *Attributes* and *Methods*. Yet does this kind of modelling create abstractions that reflect concepts of the real world completely and correctly?

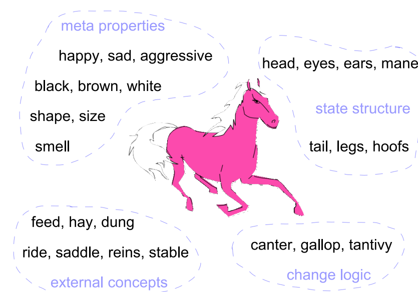

Fig. 5. Concept of a Horse

The model of a *Horse* shall serve as example to investigate this further. Figure 5 shows a number of terms commonly used to create a model of a horse. Most importantly, there are structural observations describing the horse as concept consisting of parts like *Head*, *Legs* or *Hoofs*. Secondly, there are properties like the horse's *Colour*, *Shape* or *Size*. Thirdly, there are terms describing a horse's actions like its *Movement* or *Eating*, that change a horse's position and/ or state. Finally, there are a number of terms like *Hay* or *Saddle* associating concepts related to the horse.

One might suggest to model properties like the position, size or colour of a horse's leg as *Part* of that leg. In fact, this is how classical programming approaches its solutions. In OOP, one would probably use a class representing the leg and an attribute standing for the leg's colour. However, when following the modelling principles of human thinking (see [16]), this is *not* correct!

It is true that in everyday language, one tends to say *A horse leg* has a *colour*. Unfortunately, this leads to the wrong assumption that a leg were made of a colour. But this is not the case. A leg does not *consist* of a colour in the hierarchical meaning of a whole consisting of parts. The colour is rather property information *about* the leg. It seems there is no correct expression in natural (English) language stating the property of something. The *IS-A* verbalisation is used to express that the leg belongs to a special category of items, for example: *A leg is a body element.* The *HAS-A* formulation is used to express that a leg as whole consists of smaller parts, for example: *A leg has a knee and it has a hoof.* But which formulation expresses a property? Well, perhaps it would be best to say: *A leg IS-OF a colour.*

The CYBOP knowledge schema described later in this article (section 3.2) distinguishes structural whole-part- from meta information. Actions (like the gallop of a horse) causing change in the model or its environment are called *Logic*, since they follow certain rules.

3 Reflexions on Concepts

Although many of the ideas and solutions presented here, in a *bottom-up* manner, stem from writing source code in practice (following the *Constructive Development* method of research as announced in section 1.4), the overall approach and explanation of results follow a *top-down* path. High-level concepts are considered first, before moving on to an implementation and proof in practice. Because of the steady comparison to principles of nature and other sciences, this approach is called *cybernetics-oriented*, as explained in section 1.3. Figure 6 shows in which order the elements of CYBOP will be considered.

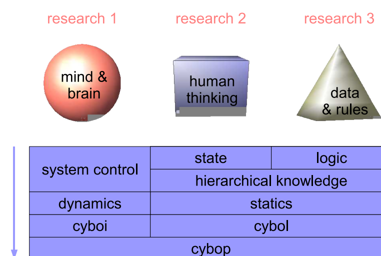

Fig. 6. Consideration of CYBOP

A first observation, when looking at human beings from a philosophical perspective, is the separation of *Mind* and *Brain* (Body). Accordingly, CYBOP treats computers as *Systems* owning and processing *Knowledge*. This is not unlike the idea of *Agent* systems owning a *Knowledge Base* [26, 23]. All abstract knowledge that humans make up belongs to their mind. The brain is merely a physical carrier of knowledge. Similarly, there are actually two kinds of software: one representing *passive* knowledge and the other *actively* controlling a system's hardware.

Secondly, attention is payed to the concepts of *Human Thinking* [16], as investigated by psychology. Through their application, knowledge becomes *hierarchical*. Moreover, this work tried to embed knowledge models in an environment of *Dimensions*, as known from physics. Every model keeps a number of *Meta Information* about its parts. *Positions* in space or time are one such example.

Thirdly, *State*- gets distinguished from *Logic* knowledge. It is known from neurological research that the human brain has special communication regions that, simply spoken, do nothing else than translating data, i.e. an input- into an output *State*, according to rules of *Logic*. Systems theory uses similar abstractions. When talking about states, this work means a composed *Set* of states.

In CYBOP (figure 6), all knowledge (states and logic), belongs to a system's *Statics*, and is described by CYBOL language templates (section 4.1). The processing of knowledge at runtime, to control a system, is *Dynamics* and happens in the CYBOI interpreter (section 4.2).

3.1 Statics and Dynamics

Of the many scientific fields that have been touched and delivered design ideas for CYBOP, only few can be elaborated in this article, due to the limited space.

Code Reduction In his book *Programming Pearls* [4, page 128], Jon Bentley demonstrates *Code Reduction* on the following graphics program example:

```
for i = [17, 43] set(i, 68)
for i = [18, 42] set(i, 69)
for j = [81, 91] set(30, j)
for j = [82, 92] set(31, j)
```

He suggests to replace the *set* procedures that switch a *Picture Element* (Pixel) with suitable functions for drawing horizontal and vertical lines:

```
hor(17, 43, 68)
hor(18, 42, 69)
vert(81, 91, 30)
vert(82, 92, 31)
```

This code, finally, gets reduced to pure data stored in an array:

```
h 17 43 68
h 18 42 69
v 81 91 30
v 82 92 31
```

The data can be read by an interpreter program which knows about their meaning.

Bentley's example shows in a nice way how knowledge can be extracted from program source code. The graphic application's actual data are represented by the values in the array above. All other functionality accessing and manipulating Pixels directly does belong to system control and remains in the interpreter program. Section 4.2 will introduce an interpreter that is able to read and handle *general* knowledge, only on a much larger scale.

Base- and Meta Level Reflective techniques as described in [19] make use of one so-called *Base Level* and one or more *Meta Levels*. The reason for splitting a system's architecture in this way is the hope to be able to move rather general *System Functionality* into a meta level, while leaving domain-specific *Application Functionality* in the base level. (Well, in his book *Analysis Patterns – Reusable Object Models* [12], Fowler used meta levels to model general classes containing not exclusively system- but also domain-specific functionality.) The conflicts a design decision of that kind can bring with were described in [19], which – above all – criticised the bidirectional dependencies.

However, what the proposition of reflective software patterns shows, is the existence of a wish among software developers, to separate general system- from more specific application functionality. And nature does exactly that. Yet while reflective mechanisms use the same implementation techniques for system- as well as for application-specific functionality, nature always treats passive knowledge strictly separate from active system control. The best example therefor is the biological cell division, where passive genetic information situated in a *Desoxy Ribo Nucleic Acid* (DNA) is read and transmitted into proteins by an active mechanism involving *Ribo Nucleic Acid* (RNA) molecules [16]. Bidirectional dependencies do not exist between the both.

Application and Domain Over the years, it has turned out to be helpful in software design, to separate *Domain Knowledge* from *Application Functionality*. In one-or-another form, the architectural software patterns [19] *Layers*, *Domain Model* and *Model View Controller* (MVC) all suggest to apply this principle.

The *Tools & Materials* approach [35] talks of *active* applications (tools) working on *passive* domain data (material). And also *System Family Engineering* [7] bases on a separate treatment of domain and application, in form of *Domain Engineering* (DE) and *Application Engineering* (AE).

An often neglected fact of these approaches is that not only the domain, but also the application contains important business knowledge (figure 7). The *User Interface* (UI), for example, is tailored for a specific business domain. And the logic behind, if not contained in the UI itself, is often put in a *Controller* which belongs to the application−, not the domain layer.

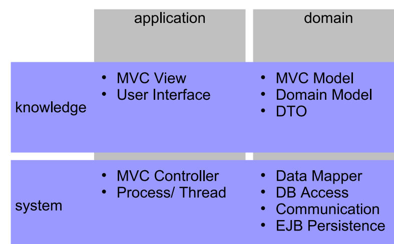

Fig. 7. Different Knowledge Separations

Similarly, the domain often contains functionality which actually does belong into the application process: *Database* (DB) access is handled by help of patterns like the *Data Mapper* [19], in which the mapper
objects contain *Structured Query Language* (SQL) code to connect to a *Database*

Management System (DBMS); *Enterprise Java Beans* (EJB), which should better be pure domain objects, imitate a *Middleware* providing persistence- or communication mechanisms, which originally have nothing to do with the business knowledge they contain.

It is precisely this *Mixup* of responsibilities between an application system and its domain knowledge, that leads to multiple inter-dependencies and hence unflexibility within a system. Instead, a separation should be made between active *System Control* and passive *Knowledge*. A UI's appearance would then be treated as domain knowledge, just as the logic of the functions called through it. A data mapper would be transformed into a simple *Translator* – similar to a *Data Transfer Object* (DTO) [19] – that knows how to convert data from one domain model into another; its DBMS access functionality, however, would be extracted and put into the application system. Monstrosities like EJBs would likewise be opened up and parted into their actual domain knowledge, and all other mechanisms around – the latter being moved into the application system.

To sum up this thought: The essential realisation here is that hardware-close mechanisms like the ones necessary for data input/ output (i/o), enabling inter-system communication, should be handled in an active application system layer which was started as process on a computer, and *not* be merged with pure, passive domain knowledge. User interfaces and application logic which are traditionally held in controller objects of the application layer, as well as further business data models, should rather belong to a high-level knowledge layer.

Platform Specific and -Independent The *Model Driven Architecture* (MDA) [25] took a first step into the right direction, by distinguishing *Platform Independent Models* (PIM), that is domain- and application logic, and *Platform Specific Models* (PSM), that is implementation technology. It encourages the use of automated tools for defining and transforming these models.

While the definition, organisation and management of architectures (PIM) mostly happen in the analysis- and design phase of a *Software Engineering Process* (SEP) (section 2.1), the generation of source code (PSM) can be assigned to the implementation phase. The approach still has weaknesses, and tools which can truly generate running systems are rare or not existent, at least to what concerns more complex software systems – not to talk of the so-called *Roundtrip Engineering*, which is managed by even less tools.

Nevertheless, the trend clearly goes towards more model-centric approaches. The aim of this work was to supply domain experts and application developers with a *Model Only* technology, allowing to create application systems that do *not* have to be transformed into classical implementation code any longer, whereby the SEP abstraction gap number *2* (figure 3) could be closed conclusively. The knowledge schema introduced in section 3.2 is a necessary prerequisite therefor.

Data Garden Now, if a distinction of high-level knowledge from low-level system control software is considered to be useful, the next question must be: *How, that is in which form, best to store knowledge in a system?*

One possible structure called *Data Garden* [20] was proposed by Wau Holland of the *Chaos Computer Club* (CCC). Although being a non-academic organisation, his ideas on knowledge modelling are interesting to this work. He dreamt of whole *Forests*, *Parks* or – as the name says – *Gardens* of *Knowledge Trees* and *Data Bushes* (figure 8).

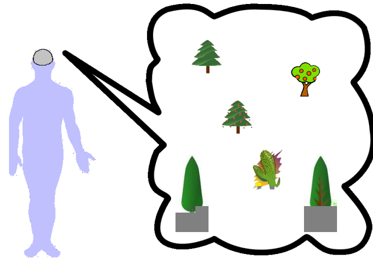

Fig. 8. Data Garden

The interpreter (section 4.2) created in the work described in this article stores all its knowledge in *one single* tree, whose root node it references. The single concepts (data bushes) are represented by branches of that knowledge tree.

3.2 Knowledge Schema

Human beings have a brain which they use to think, in other words to build up a mind. While the former exists in the *Real World*, the latter is constructed as a subjective *Virtual World*. All people do think, all the time, even not knowing that they do. One would therefore guess that the act of *Thinking* is a most common one, familiar to anybody. But judging from the enormous research effort in sciences dealing with it, the *Principles* behind thinking are not that easy to grasp.

Schema A theoretical *Model* is an abstract clip of the real world, and exists in the human mind. Another common word for *Model* is *Concept*. It is the subsumption of *Item*, *Category* and *Compound*, resulting from three activities of abstraction: *Discrimination*, *Categorisation* and *Composition*, as explained in [16]. Each model *knows* about the parts it consists of.

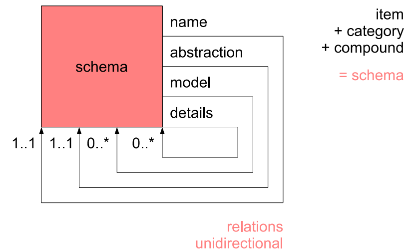

Fig. 9. Knowledge Schema

Yet what does this knowledge of a compound model (whole) about its parts imply? Software developers call knowledge *about* something *Meta Information*. Figure 9 illustrates a *Schema* (structure) with four kinds of meta information in a whole-part relation.

An obvious way is to give each part a unique *Name* for identification. Secondly, a compound needs to know about the *Model* of each part since a part may itself be seen as compound that needs to know about its parts. The distinction of the several kinds of models, in other words the kind of *Abstrac-*

tion (compound, term, number etc.) of a model is the third kind of information a compound needs to know about its parts. It is comparable to a *Type* in classical system programming languages. All further kinds of meta information are summed up by a fourth relation which is called *Details*.

Double Hierarchy Finally, what makes up the character of a model (in the understanding of the human mind) is a combination of two hierarchies: the *Parts* it consists of, together with *Meta Information* about it.

Most properties of a molecule in *Chemistry*, for example, are determined by the number and arrangement of its atoms. *Hydrogen* (H_2) becomes *Water* (H_2O) (with a totally different character) when just one *Oxygen* (O) atom is added per hydrogen molecule.

The kinds of meta information discussed in [16] were also called *Dimensions* or *Conceptual Interaction* between a *Whole* and its *Parts*. They may represent very different properties and be constrained to certain values- or areas of validity.

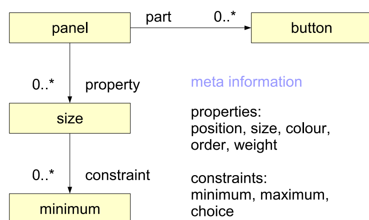

Fig. 10. Double Hierarchy (Parts | Meta)

Figure 10 illustrates the *Double Hierarchy* here spoken of. A graphical panel was chosen as example model. It consists of smaller parts, among them being a number of buttons. Altogether they form the *Part Hierarchy*. On the other hand, there are properties like the size, position or colour of the buttons, which are neither part of the panel, nor of the buttons themselves; they are information *about* the buttons and form an own *Meta Hierarchy*. To the latter do also belong constraints like the minimum size of a button or a possible choice of colours for it. *Properties* are (meta) information about a *Part*; *Constraints* about a *Property*.

Container Unification Section 1.2 mentioned container inheritance as one problem of current software. Due to polymorphism, it may cause unpredictable behaviour possibly leading to *falsified* container contents [24]. The previous sections introduced a knowledge schema which they claimed to be *general*. But that also means that all kinds of containers must be representable by the suggested schema. But why are there so many different kinds of containers? What actually is a container?

It is a concept expressing that some model *contains* some other model(s). Types of containers are, for example: *Collections* (Array, Vector, Stack, Set, List), *Maps* (Hash Map, Hash Table) and the *Tree*. They all are containers. What differs, is just the meta information they store about their elements. A list, for example, holds position information about each of its elements. A map relates the name of an element to its model (1:1). A tree links one model to many others (1:n).

But does the different meta information a container holds about its elements justify the existence of different container models? If a knowledge schema was general enough to represent a container structure on one hand, and to express different kinds of meta information on the other, it might be able to behave like any of the known container types.

The schema proposed in this work claims to be this kind of knowledge schema. It has a container structure by default, and can thus hold many parts in a *Tree*-like manner. It holds standard meta information about its parts: their *Name*, *Model*, kind of *Abstraction* and further meta information called *Details* – and is therefore able to link the name of an element to its model, in a *Map*-like manner. To the additional meta information (details) may belong the *Position* of an element within its model, in a *List*-like manner. A *Table* structure can be represented as well, by splitting it into a hierarchical (tree-like) representation, as known from markup languages like the *Hypertext Markup Language* (HTML).

Section 4.1 will introduce a language capable of expressing all aspects of the knowledge schema as proposed here.

Universal Memory Structure To better explain the differences between traditional- and cybernetics-oriented design models, an example shall help. (A first one was given in section 2.3, which showed modelling mistakes at the concept of a horse.) Figure 11 illustrates design-time structures in the upper half, and runtime structures in the lower. Using *Structured- and Procedural Programming* (SPP) or *Object Oriented Programming* (OOP), a developer would design a model as shown on the upper left-hand side in the figure. (The fact that OOP also offers inheritance relations and OOP classes do own methods in addition to attributes, while SPP structures do not, is of minor importance here.) At runtime, exactly that model would be applied to structure instances and their relations accordingly, as shown on the lower left-hand side in the figure.

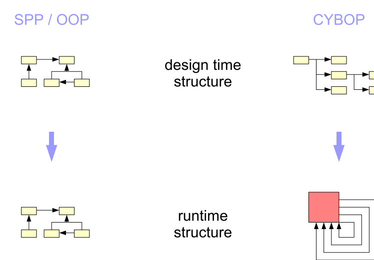

Fig. 11. Universal Memory Structure

Not so in *Cybernetics Oriented Programming* (CYBOP). Knowledge templates as created at design time do always have a hierarchical structure, as shown on the upper right-hand side in the figure. They include *Whole-Part*- as well as *Meta Hierarchies* (the latter neglected in the figure). At runtime, these templates get cloned by creating models that follow the structure of the CYBOP *Knowledge Schema*, as shown on the lower right-hand side in the figure. While SPP/ OOP rely on a variety of different structures to store knowledge in memory, CYBOP uses one *Universal Memory Structure* (knowledge schema) that, so to say, merges traditional structures like different kinds of *Containers*, *Class* and *Record/Struct*. Even algo-

rithmic structures (logic) traditionally stored in a *Procedure* are covered by this knowledge schema. More on state and logic in the following section.

The advantages are obvious. Data available in a unified structure are easier to process. Dependencies of the knowledge schema are defined clearly and remain the same for all applications, so that domain/ application knowledge becomes independent from the underlying system control software. Global data access and bidirectional dependencies are not necessary anymore, since every knowledge model can be accessed along well-defined paths within the knowledge hierarchy. Byte code manipulation and similar tricks and workarounds might finally belong to the past.

3.3 State and Logic

This section investigates how classical software system design handles *State-* and *Logic Knowledge* and which role they play in system communication.

Interacting Systems Figure 12 shows a simplified example *Information Technology* (IT) environment (*Physical Architecture*), containing many interacting systems: server and client, local and remote, human and artificial. In (object oriented) software design, special patterns are used to architect a system such that it is able to communicate with other systems across various mechanisms (*Logical Architecture*). To these patterns count the *Data Mapper*, *Data Transfer Object* and *Model View Controller* [13].

Although software development has become a lot easier in the last decades, it is still a big effort that should not be underestimated. One thing that application developers have to care about

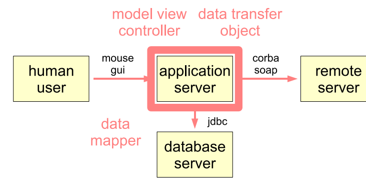

Fig. 12. IT Environment with Patterns

much of their time is the *Conversion* between various kinds of (communication) models that a system has:

- Frontend (with Human User)
- Backend (with Data Source)
- Remote (with Server)
- Domain (with own Knowledge)

The different mechanisms and patterns that have to be considered for such model conversion often need to be implemented repeatedly, for each new application. Some trials to unify all backend communication in a common *Persistence Layer* exist [1], but are remote- and frontend communication seldom considered in a comparable way. Obviously, no current effort treats the frontend as just another communication model that has to be *sent* to the human user as just another system.

Pattern Simplification The three communication patterns mentioned before had already been reinvestigated for commonalities in [18], which also embedded them into the classical model of logical system layers (figure 13). For all kinds of communication, there is a:

- *System* (Human User, Database, Remote Server)

- *Model* (View, ERM, DTO)
- *Translator* (Controller/ View Assembler, Data Mapper, DTO Assembler)

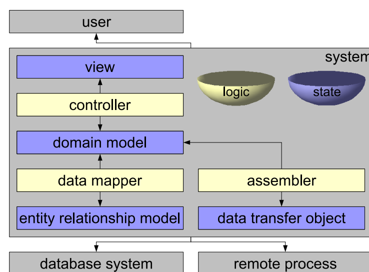

Fig. 13. Simplified Patterns in Layers

All models represent certain states; all translators contain logic for converting one state into another; all systems host their own, specific pool of state- and logic knowledge. Realising this, a much clearer view on software architectures can be retrieved.

Because domain models differ between systems, each system needs its own translator models. Only communication models need to be agreed upon between systems; they need to be understood by both communication partners.

Communication Model Systems (alive or not) never communicate directly, but always across the detour of an external (transient or persistent) *Medium*. This makes it necessary to use special *Communication Models*, since nearly always, only *parts* of a complete *Domain Model* want to be exchanged. The use of communication (transfer) models again, entails the use of model *Translators*. Sowa [30] writes in his book *Knowledge Representation*:

> In computer science, there is no end to the number of specialized notations. Besides the hundreds of programming languages, there are diagrams for circuits, flowcharts, parse trees, game trees, Petri nets, PERT charts, neural networks, design languages, and novel notations that are invented whenever two programmers work out ideas at the blackboard. Musical notation . . . is an example of a complex language that is both precise and human factored. As long as the mapping rules are defined, all of these notations can be automatically translated to or from logic.

Although he does not talk of *Domain-* and *Communication Models*, but of *Notations*, Sowa obviously means the same: Any kind of abstract model can be translated into any other kind, as long as the translation *Rules* are defined. Model *Translators* are able to map domain model data to transfer model data. Depending on which communication style is used, different translators with different rules need to be applied.

Figure 14 shows a number of possible model translators, for a: *Textual User Interface* (TUI), *Graphical User Interface* (GUI) and *Web User Interface* (WUI) as well as for the German standard file format for exchanging medical data called *x Daten Träger* (xDT), the *Healthcare Xchange Protocol* (HXP) and *Health Level Seven* (HL7)'s exchange format called *Clinical Document Architecture* (CDA).

Many application systems have exactly one domain model but transfer

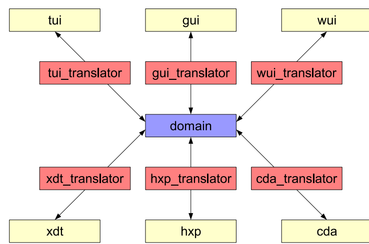

Fig. 14. Different Model Translators

models of arbitrary type should be addable anytime. Translators only know how to translate between the domain model and a special transfer model, of course in both directions. *Direct* translation between transfer models is an exception; it is possible but better done *via* the domain model.

The type of transfer model is independent from the communication mechanism used. The usage of a *Graphical User Interface* (GUI) model, for example, is not necessarily limited to human-computer interaction. It could very well be used for data transfer between remote computers, as long as both systems know how to translate that model.

Logic manipulates State According to the observations made in the work described in this article, there are two kinds of knowledge: *States* and *Logic*. While the former may be placed in a spatial dimension, the latter is processed as sequence over time. Often, logic is labelled *dynamic* behaviour – but only the *execution* of a rule of logic is dynamic, *not* the rule itself (*static*).

Rules of logic translate input- into output states. What characterises a system is how it applies logic knowledge to translate state knowledge [15]. Yet how to imagine a knowledge model consisting of state- as well as logic parts? Following an example. The famous *Model View Controller* (MVC) pattern was extended by the *Hierarchical MVC* (HMVC) pattern towards a hierarchy of *MVC Triads* [6]. The omnipresence of hierarchies in the MVC was demonstrated in [17].

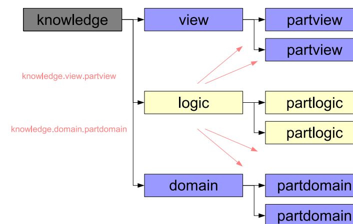

Fig. 15. Logic manipulating States

Figure 15 shows the three parts: *Domain* (Model), *View* and *Logic* (Controller) of an (adapted) MVC pattern as independent branches of one common knowledge tree, as existent at system runtime in memory. Each of them represents a concept on its own. The logic model, however, is allowed to access and change the view- and domain model; it is able to link different knowledge models. But view- and domain model, representing states, are not allowed to manipulate logic. In other words: The dependencies between logic- and state models are *unidirectional*.

An innovation is that logic knowledge gets manipulatable. A logic model (algorithm) cannot only access and change

state-, but also logic models, even itself! Because models modified in that manner can be made persistent in form of CYBOL knowledge templates (section 4.1), and be reloaded the next time an application starts, this may be seen as a kind of *Meta Programming*, which [9] defines as: *the writing of programs that write or manipulate other programs (or themselves) as their data.*

The clear separation of states and logic into discrete models avoids unwanted dependencies as caused by the bundling of attributes and methods in OOP. All that would be needed to make a CYBOP system work with new state models, is the corresponding translation logic. Translators [18] simplify architectures and unify communication.

```
<!ELEMENT model (part*)>
<!ELEMENT part (property*)>
<!ELEMENT property (constraint*)>
<!ELEMENT constraint EMPTY>

<!ATTLIST part
    name CDATA #REQUIRED
    channel CDATA #REQUIRED
    abstraction CDATA #REQUIRED
    model CDATA #REQUIRED>

<!ATTLIST property
    name CDATA #REQUIRED
    channel CDATA #REQUIRED
    abstraction CDATA #REQUIRED
    model CDATA #REQUIRED>

<!ATTLIST constraint
    name CDATA #REQUIRED
    channel CDATA #REQUIRED
    abstraction CDATA #REQUIRED
    model CDATA #REQUIRED>
```

Fig. 16. CYBOL DTD

4 Practical Proof

The proof of operatability for the new concepts is given by the *Cybernetics Oriented Language* (CYBOL), defined according to the principles of abstraction worked out before, and by the *Cybernetics Oriented Interpreter* (CYBOI), a knowledge processing system. In addition, a prototype application called *Res Medicinae* [27] was implemented in CYBOL.

4.1 CYBOL

Document Type Definition Since CYBOL is based on the *Extensible Markup Language* (XML), a *Document Type Definition* (DTD) can be given (figure 16).

One can recognise the purely hierarchical structure as described by the CYBOP knowledge schema (section 3.2). The three elements *part*, *property* and *constraint* have the same list of required attributes.

Hello World The well-known *Hello, World!* program printing just two words shall be given as minimal example application. It consists of only two operations: *send* and *exit*. The string message to be displayed on screen is handed over as *property* to the *send* operation, before the *exit* shuts down the system:

```
<model>
    <part name="send_model_to_output"
        channel="inline"
        abstraction="operation"
        model="send">
        <property name="language"
            channel="inline"
            abstraction="string"
            model="tui"/>
        <property name="receiver"
            channel="inline"
            abstraction="string"
            model="user"/>
        <property name="message"
            channel="inline"
            abstraction="string"
            model="Hello, World!"/>
    </part>
    <part name="exit_application"
        channel="inline"
        abstraction="operation"
        model="exit"/>
</model>
```

Container Mapping State-of-the-art programming languages offer a number of different container types, partly based on each other through inheritance. Sections 1.2 and 3.2 of this work mentioned *Container Inheritance* as one reason for falsified program results.

Container	Knowledge Template
Tree	Hierarchical *whole-part* structure
Table	Like a Tree, as hierarchy consisting of rows which consist of columns
Map	Parts have a *name* (key) and a *model* (value)
List	Parts may have a *position* property
Vector	A *model* attribute may hold comma-separated values; a template holds a number of parts (dynamically changeable)
Array	Like a Vector; characters are interpreted as *string*

Table 1. Containers in CYBOL

Section 3.2 introduced a *Knowledge Schema* which represents each item as *Hierarchy* by default, the result being that different types of containers are not needed any longer. But how are the different kinds of container behaviour implemented in CYBOL? Table 1 gives an answer.

4.2 CYBOI

The pure existence of proper knowledge does not suffice to create an improved kind of software system, within a slimmer software development process. The system needs to know how to *handle* knowledge, at runtime. The criticism is twofold, since traditionally:

1. Operating systems don't have sufficient knowledge handling capabilities
2. Applications contain too much low-level system control functionality

This is changed when using CYBOI. As active interpreter encapsulating system-level functionality, it handles knowledge provided in form of passive CYBOL templates. In CYBOP systems, all compound knowledge models have the same type structure (schema). Since they do not differ, they can be manipulated in the same manner.

Overall Placement Considering an overall computer system architecture, *CYBOI* is situated between the application knowledge existing in form of *CYBOL* templates and the *Hardware* controlled by an *Operating System* (OS) (figure 17). CYBOI can thus also be called a *Knowledge-Hardware-Interface* (synonymous with *Mind-Brain-Interface*).

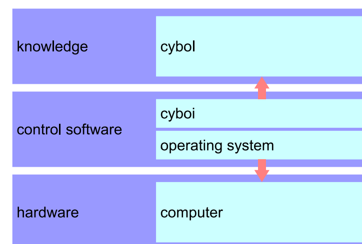

Fig. 17. Knowledge – Hardware Link

There are analogies to other systems run by language interpretation.

Criterion	**Java World**	**CYBOP World**
Theory	OOP in Java	CYBOP
Language	Java	CYBOL
Interpreter	Java VM	CYBOI

Table 2. Java-/ CYBOP World Analogies

Table 2 shows those between the *Java-* and *CYBOP* world. Both are based on a programming theory, have a language and interpreter. A theoretical model of a computer hardware- or -software system may be called an *Abstract Computer* or *Abstract Machine* [9]. If being implemented as software simulation, or if containing an interpreter, it is called a *Virtual Machine* (VM). Kernighan and Pike write in their book *Practice of Programming* [22]:

> Virtual machines are a wonderful, old idea, that latterly, through Java and the *Java Virtual Machine* (JVM), came into fashion again. They are a simple possibility to gain portable and efficient program code, which can be written in a higher programming language.

In that sense, CYBOI is certainly a VM. It provides low-level, platform-dependent system functionality, close to the OS, together with a unified knowledge schema (section 3.2) which allows CYBOL applications to be truly portable, well extensible and easier to program, because developers need to concentrate on domain knowledge only. Since CYBOI interprets CYBOL sources *live* at system runtime, without the need for previous compilation (as in Java), changes to CYBOL sources get into effect right away, without restarting the system.

Architecture To what concerns its inner architecture, there are two basic structures underlying CYBOI:

1. *Knowledge Container:* An array-based structure usable for storing static knowledge in form of primitive- and compound models, and capable of representing a map, collection, list and tree
2. *Signal Checker:* A loop-based structure usable for dynamically reading signals from a queue, and capable of processing them after their priority, in a special handler

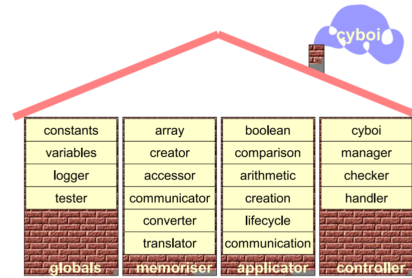

Fig. 18. CYBOI Architecture

All modules, into which CYBOI is subdivided, are built around these two core structures. Not unlike John von Neumann's model of a computing machine [32], which distinguishes *Memory*, *Control Unit*, *Algorithmic Logic Unit* (ALU) and *Input/ Output* (i/o), CYBOI's modules are grouped into four architectural parts, as illustrated in figure 18. These have the following functionality:

– *Memoriser:* data creation, -destruction and -access (after Neumann, it con-

tains not only data, but also the operations that are applied to them)
- *Controller:* lifecycle management, signal handling, i/o filters
- *Applicator:* operation application (comparison, logic, arithmetic and more)
- *Globals:* basic constants and variables, as well as a logger

The i/o data handling is not separated out here (as opposed to von Neumann's model); it is managed by the controller modules. The i/o data themselves, representing states, are stored in memory.

Functionality Figure 19 shows three main parts of CYBOI. (The *Globals* package is neglectable for the following explanations, since it contains static constants and variables that are *omnipresent*.) The *Controller* manages system startup, shutdown and the handling of signals during its runtime; the system uses just one central signal checking loop. The *Memoriser* provides memory structures (to store knowledge) and procedures to access these. Logic knowledge is processed in the *Applicator*.

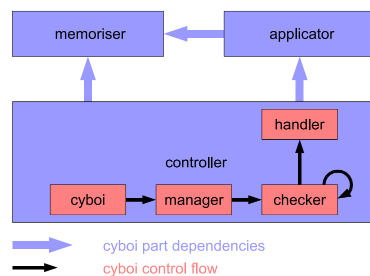

Fig. 19. Dependencies and Control Flow

4.3 Res Medicinae

Project Background The – somewhat idealistic – aim initially was to create the prototype of a *Hospital Information System* (HIS). Due to the clearly too high-set aims, this was later revised so that the focus of the prototype became a standard *Practice Management System* (PMS) with an *Electronic Health Record* (EHR) as its core. Several technology changes during the progress of this work and the lack in time required to also revise this aim, so that now the final prototype consists of just the (rudimentary) address management module of the planned EHR application. It is written in CYBOL and executable by CYBOI.

First Trial An early trial of a *Res Medicinae* module was *Record*, an application for EHR management. It was a standard Java-based system and had a *Graphical User Interface* (GUI). Its classical architecture made use of many software patterns and was shared into the parts *Domain Model*, *Graphical View* and *Controller*, as proposed by the equally named pattern, abbreviated *MVC*.

Later prototypes extended that architecture by applying the CYBOP concept of *Composition*. In a first step, the *Hierarchical MVC* (HMVC) pattern was used to replace the MVC pattern, resulting in nested *Controllers* and *Views*. Afterwards, the principle of *Hierarchy* was applied in general, also to *Domain Models* and to as many other parts as possible.

Classes as known from *Object Oriented Programming* (OOP) do not represent dynamically extensible containers but have a static structure with a fixed number of attributes. In other words,

the *Hierarchy* as concept is not inherent in OOP types. Yet abstract models as humans build them in their minds are always based on hierarchies (section 3). A programming language which does not consider this, does not allow users to make full use of their modelling potential.

To eliminate this flaw and implement a hierarchical structure in the Java prototype, a top-most super class named *cybop.Model* had to be introduced. It represented a container that had the capability to reference itself – in other words a *Tree Structure*. As such, it offered *set*, *get* and *remove* methods for its elements. Since these access methods were inherited, sub classes did not have to implement their own (for each attribute) anymore, which saved hundreds of lines of source code.

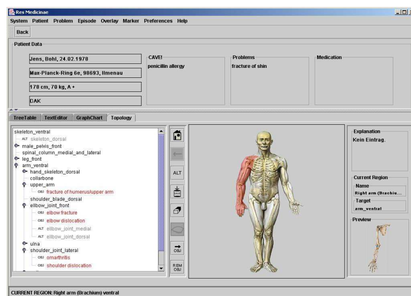

Fig. 20. Topological Documentation

One of these advanced modules, to give an example, was responsible for clinical documentation [17], which it supported graphically, in form of *Topological Documentation* (figure 20). And, of course, it could also manage and store patient data, in XML files.

Knowledge Separation In the case of the first prototypes, one could still speak of true *Implementation*, because design models had to be transferred into another form of abstract model: the Java programming language source code. Not so in later versions of *Res Medicinae*.

While the early prototypes represented the classical mix of domain knowledge and low-level system instructions, that was eliminated later. All knowledge got *extracted* and was put into special configuration files, in *CYBOL* format (section 4.1). Henceforth, these contained not only settings like font size or colour, as known from standard applications, but the *whole* domain knowledge, including user interface- and workflow structures.

Following the explanations of section 3.3, the *static* knowledge was shared into different models, some representing *state*-, and others *logic* knowledge. This was very much opposed to the earlier Java implementations whose classes bundled attributes and methods.

Without the knowledge, the remaining program code looked pretty much like a skeleton of basic system functionality. Serving as hardware interface, it concentrated memory- and signal handling in one place – exactly those things which section 3.1 called *Dynamics*. Additionally, that remaining system had the ability to interpret knowledge, which is why it was called *CYBOI* (interpreter). One could, in some way, compare it with what the *Java Virtual Machine* (JVM) is for Java, only that CYBOI processed knowledge given in form of CYBOL templates, which look different than Java source code.

CYBOI needed an *XML Parser* in order to be able to read the knowledge contained in CYBOL files. The decision

here fell on Apache's *Xerces* [28], because one of its versions is implemented in Java.

Reimplementation The architecture-advanced prototype of the *Record* module had *much* less functionality than earlier ones, in fact not much more than starting a graphical frame with menu bar and exiting the application again. This was so, because yet before all domain knowledge could be extracted into CYBOL, another issue turned up:

CYBOP modelling concepts like *Itemisation* or *Composition* are an integral part of the CYBOL knowledge representation language. Other concepts like the *Bundling* of attributes and methods, property- and container *Inheritance*, as known from *Object Oriented Programming* (OOP), were considered unfavourable (section 2) and neither to be used in CYBOL, nor in the CYBOI interpreter. Consequently, OOP languages like Java or C++ were not suitable for CYBOI any longer. A slim and fast language, close to hardware and fast in processing CYBOL was needed.

Having such requirements, one of the first candidates coming to mind was the *C* programming language. It is *high-level* enough to permit fast programming and *low-level* enough to connect efficiently to hardware or an *Operating System* (OS). Many OS are written in C themselves, anyway. CYBOI was therefore reimplemented in C, which hasn't changed since. What has changed and is changing all the time is its functionality, an overview of which was given in section 4.2.

One problem that had to be solved was *Graphical User Interface* (GUI) handling. While the Java-implemented CYBOI could make use of the *Abstract Windowing Toolkit* (AWT)/ Swing, the C-implemented CYBOI did not have such functionality at first. Toolkit candidates like *Qt* [34] or *wxWindows* [29], being implemented in C++, were out. Other GUI frameworks like the *Gimp Toolkit* (GTK) [31], written in C, were considered cumbersome to cope with so that finally, the decision was taken to use low-level graphics drawing routines. For CYBOI, being developed on a *GNU/Linux* OS [33], that meant using *XFree86's* [8] *X-Library* (Xlib) functionality directly. The necessary effort for transforming hierarchical CYBOL models into GTK- or other toolkit structures was estimated to be equal or even higher than translating them into Xlib functionality right away. At the time of writing this, implementation is in progress but not completed.

Similar implementations are necessary for *Textual User Interfaces* (TUI), *Web User Interfaces* (WUI) and *Socket Communication Mechanisms*, the latter two being already finished in a first version. Further development activities may for instance enable CYBOI to run on other platforms and integrate more hardware-driving functionality, to get independent from underlying OS.

While the CYBOL specification can be considered quite mature, CYBOI, as could be seen, will need plenty of extensions and additions in future, in order to leave its prototype stage and become fully usable.

Module Modelling When CYBOI had become more stable (besides the extensions that were – and are – frequently implemented, development could focus on the actual application again. From now on, *Res Medicinae* modules only had to be *modelled* in CYBOL, but no

longer had to be *coded* in a programming language. The designed state- and logic knowledge, existing in form of CYBOL templates, already represented the complete application; no further implementation phase was needed.

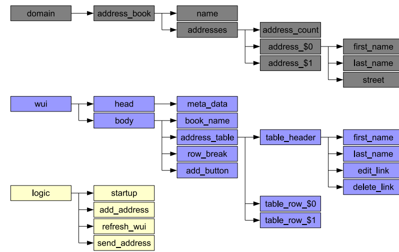

Fig. 21. ResAdmin Knowledge Models

Due to the tremendous complexity of an *Electronic Health Record* (EHR), only a very small part of its data could be considered for the application prototype. Administrative data like a person's name or address are standard information found in all EHRs. A corresponding module named *ResAdmin* [21] was therefore elected to be realised first. Its models belong to three categories: *Domain*, *Web User Interface* (WUI) and *Logic* (figure 21).

The addresses contained in the *domain* branch of the knowledge tree are manipulated across *Hyper Text Markup Language* (HTML) *User Interface* (UI) models belonging to the *web* branch of that same tree. An example structure of a knowledge tree was shown in figure 15. Every action model that a user can trigger through the WUI exists as part of the *logic* branch of the knowledge tree.

Independently of what kind of knowledge model (state or logic) was created, ontological principles were strictly followed. Most importantly, relations within a hierarchical model were always *unidirectional*, that is from a *Whole-* to its *Part* models, but never the other way around. Additionally, however, logic models may reference and access runtime state models.

Some of the logic models represent *Translators* (compare section 3.3). They extract address information residing in the domain- and copy them to the web model, which is afterwards sent to the human user as communication partner. This principle holds true for the communication between application systems, only that then other than web models are used as communication format. The vision to make all communication channels really *transparent* and easy to handle for the user now seems to be coming true.

5 Related Work

There exists a plethora of – partly *Open Source Software* (OSS) – projects promoting XML-based programming, using the *Extensible Markup Language* (XML) as replacement for a programming language. Many of these in fact follow the principles of *Structured- and Procedural Programming* (SPP) or *Object Oriented Programming* (OOP) with just another syntax, and try to map the corresponding constructs to XML. They are far away from the system control/ knowledge separation that CYBOP wants to reach.

Further, XML is often used for specifying user interfaces or workflows, the latter mostly in commercial systems. These approaches come closer to what

CYBOP does with its language CYBOL, only that CYBOL can express not only user interfaces and workflows, but also domain knowledge and algorithms.

The idea of separating system control and knowledge is used in the *OpenEHR* project [3], which inspired CYBOP in its beginnings. OpenEHR follows a meta model approach (which it calls *Dual Model*) that is based on Fowler's *Analysis Patterns* [12] describing a kind of ad hoc two-level modelling, using a *Knowledge Level* and *Operational Level* – as described by the *Reflection* pattern, which calls the two levels *Meta Level* and *Base Level*, respectively. The difference between the dual model approach and classical meta architectures is that the latter implement both, meta- and base level using the same technology (language). OpenEHR, on the other hand, uses so-called *Archetypes* for specifying knowledge, written in a special language. Besides obvious benefits of OpenEHR's approach in constraining domain knowledge, there are a number of weaknesses:

- mix of meta information (properties, constraints) and hierarchical whole-part structure
- incomplete domain knowledge lacking logic (algorithms/ workflows) and user interfaces
- inflexible structures due to runtime-dependency of instances from archetypes
- use of object-oriented concepts with all their limits

Although the *OpenEHR* project claims archetypes to be both: *domain-empowered* and *future-proof*, the above-mentioned issues prevent them from being so. The dual model approach in conjunction with archetypes only partly fulfills the expectations of independent and complete knowledge structures.

6 Summary and Outlook

This article tried to sum up a much larger scientific work entitled *Cybernetics Oriented Programming* (CYBOP). In particular, it reflected on knowledge modelling and its implications on software design. Traditional concepts were revised with new ideas stemming from various other scientific disciplines.

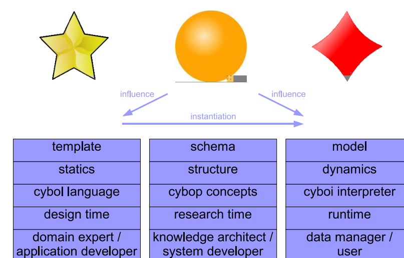

Fig. 22. Knowledge Triumvirate

The results can be reduced to one illustration: the *Knowledge Triumvirate* (figure 22). Its centrepiece is the new CYBOP knowledge *Schema* providing a structure to both, knowledge templates and -models. CYBOI *Models* are the dynamic runtime instances of static design-time CYBOL *Templates*.

Because all knowledge is stored in tree-form, application systems become much more flexible than complex class networks as known from OOP. Tree structures are easy to edit. They allow to better estimate changes caused by new requirements, because dependencies are obvious. Software maintenance gets improved, because application developers can focus on pure domain knowledge; low-level system functionality is provided by CYBOI. CYBOL applications

are therefore not only portable, but represent truely long-life systems.

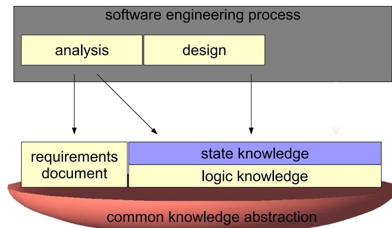

Fig. 23. Common Knowledge Abstraction

Although this work does not address the *Software Engineering Process* (SEP) directly, its results have great effect on it. Section 2.1 pointed out abstraction gaps and multiple development paradigm switches, happening during a software project's lifetime. It set out to find a *Common Knowledge Abstraction* for all phases. The results of this work help overcome *Gap 2* (figure 3). Since knowledge gets interpreted directly, the formerly needed implementation phase disappears (figure 23).

CYBOP applications are capable of communicating universally. CYBOI contains all necessary mechanisms, so that it suffices to issue a *send/ receive* operation with the corresponding language, in a CYBOL template.

Naturally, there are limits to CYBOP. It does not claim to be *the* approach for all kinds of programming problems, although it thinks to contribute suitable concepts for business application development. However, its usability for hardware-close systems with Real Time (RT) requirements is questionnable, as it cannot guarantee signal execution in time.

References

1. Scott W. Ambler. The design of a robust persistence layer for relational databases. Online White Paper, November 2000. http://www.ambysoft.com/persistenceLayer.html.
2. Federico Barbieri, Stefano Mazzocchi, and Pierpaolo Fumagalli. *Apache Jakarta Avalon Framework*. Apache Project, 2002. http://avalon.apache.org/.
3. Thomas Beale, Sam Heard, and et al. Open electronic health record (openehr) project, formerly good european/ electronic health record (gehr), April 2005. http://www.openehr.org.
4. Jon Bentley. *Perlen der Programmierkunst. Programming Pearls*. Addison-Wesley, http://www.aw.com, Boston, Muenchen, 2000.
5. Frank Buschmann, Regine Meunier, Hans Rohnert, and et al. *Pattern-orientierte Softwarearchitektur. Ein Pattern-System*. Addison-Wesley, Bonn, Boston, Muenchen, 1. korr. nachdruck 2000 edition, 1998. http://www.aw.com/.
6. Jason Cai, Ranjit Kapila, and Gaurav Pal. Hmvc: The layered pattern for developing strong client tiers. Java World Online Magazine, July 2000. http://www.javaworld.com/javaworld/jw-07-2000/.
7. Software Engineering Institute Carnegie Mellon University. Domain engineering: A model-based approach. Website containing technical Reports, 2003. http://www.sei.cmu.edu/domain-engineering/domain_engineering.html.
8. The XFree86 Developers Community. Xfree86 – the open source x window system, June 2004. http://www.xfree86.org/.

9. Collaborating contributors from around the world. Wikipedia – the free encyclopedia. Web Encyclopedia, October 2004. http://www.wikipedia.org.
10. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Principles and Techniques of Software Engineering based on automated Configuration and Fragment-based Component Models*. Addison-Wesley, Boston, Muenchen, 2000. http://www.aw.com.
11. Bernd Daene. Personal talk, September 2004.
12. Martin Fowler. *Analysis Patterns – Reusable Object Models*. Addison-Wesley, Boston, Muenchen, 1997. http://www.aw.com.
13. Martin Fowler and et al. *Patterns of Enterprise Application Architecture (Information Systems Architecture)*. Addison-Wesley, Boston, Muenchen, 2001-2002. http://www.aw.com.
14. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gang Of Four). *Design Patterns. Elements of reusable object oriented Software*. Addison-Wesley, Bonn, Boston, Muenchen, 1st edition, 1995. http://www.aw.com.
15. Christian Heller. Cybernetics oriented programming (cybop) in res medicinae. In *OSHCA Conference Online Proceedings*, Los Angeles, November 2002. Open Source Health Care Alliance (OSHCA). http://www.oshca.org/.
16. Christian Heller. Cybernetics oriented language (cybol). *IIIS Proceedings: 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004)*, V:178–185, July 2004. http://www.iiisci.org/sci2004 or http://www.cybop.net.
17. Christian Heller, Jens Bohl, Torsten Kunze, and Ilka Philippow. A flexible software architecture for presentation layers demonstrated on medical documentation with episodes and inclusion of topological report. *Journal of Free and Open Source Medical Computing (JOSMC)*, 1(26.06.2003):Article 1, June 2003. http://www.josmc.net.
18. Christian Heller, Torsten Kunze, Jens Bohl, and Ilka Philippow. A new concept for system communication. *Ontology Workshop at OOPSLA Conference*, October 2003. http://swt-www.informatik.uni-hamburg.de/conferences/oopsla2003-workshop-position-papers.html.
19. Christian Heller, Detlef Streitferdt, and Ilka Philippow. A new pattern systematics. http://www.cybop.net, March 2005.
20. Herwart (Wau) Holland-Moritz. Der datengarten. Internet Website, 2003. http://www.wauland.de/datagarden.html.
21. Rolf Holzmueller. Untersuchung der realisierungsmoeglichkeiten von cybol-webfrontends, unter verwendung von konzepten des cybernetics oriented programming (cybop). Master's thesis (diplomarbeit), Technical University of Ilmenau, Ilmenau, June 2005. http://www.cybop.net.
22. Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Boston, Muenchen, 1999.
23. Ralf Kuehnel. *Agentenbasierte Softwareentwicklung: Methode und Anwendungen*. Agenten Technologie. Addison-Wesley, Muenchen, 2001.
24. Peter Norvig. The java iaq: Infrequently answered questions. http://www.norvig.com/java-iaq.html.
25. Object Management Group (OMG). Model driven architecture (mda), March 2002. http://www.omg.org/mda/.
26. David Parks. Agent oriented programming: A practical evaluation. Web Article, May 1997. http://www.cs.berkeley.edu/ davidp/cs263/.
27. Res Medicinae Project. Res medicinae – medical information system, 1999-2004. http://www.resmedicinae.org.

28. The Apache XML Project. Xerces java parser, 2003. http://xml.apache.org/xerces2-j/index.html.
29. Julian Smart, Anthemion Software Ltd., and et al. wxwidgets (formerly: wxwindows) cross-platform native ui framework, April 2005. http://www.wxwidgets.org/.
30. John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Brooks/Cole, Thomson Learning, Pacific Grove, 2000.
31. GTK Team. Gimp toolkit (gtk), April 2005. http://www.gtk.org/.
32. SelfLinux Team. *SelfLinux – Linux-Hypertext-Tutorial.* PingoS e.V., Hamburg, 0.11.3 edition, June 2005. http://www.selflinux.org/.
33. Linus Torvalds, Alan Cox, and et al. The linux operating system kernel, 2004. http://www.kernel.org/.
34. Trolltech. Cute toolkit (qt) c++ application development framework, April 2005. http://www.trolltech.com/products/qt/index.html.
35. Heinz Zuellighoven and et al. Tools & materials approach to software-development. JWAM Open Source Project, 2004. http://www.jwam.de/engl/produkt/e_tmapproach.htm.