Reflexions on Knowledge Modelling

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

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

Abstract: This article reports about an ongoing research investigating the possibilities for applying interdisciplinary concepts to software system design. The new resulting programming philosophy is based on firstly, a 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. It solves many of the problems existing in classical programming paradigms and languages and may have the potential to replace these in the long run.
Keywords: Knowledge Abstraction, Cybernetics Oriented Programming, CYBOP, Software Design

1 Introduction

Sometimes, describing the easy things is the most difficult. And most of the time, it seems easier to copy existing concepts than to investigate new, but possibly more intuitive solutions. The work described in this document tried to question traditional concepts of software design and to correct or simplify these by applying new ideas stemming from other scientific disciplines. It thus wants to contribute to a better knowledge modelling.

The initially observed discrepancies belong to software engineering processes (abstraction gaps), to the physical architecture (misleading tiers) as well as the logical architecture (modelling mistakes) of systems. They are explained following.

1.1 Abstraction Gaps

Software has to be developed in a creative process called *Software Engineering Process* (SEP) or -*Methodology* (figure 1).

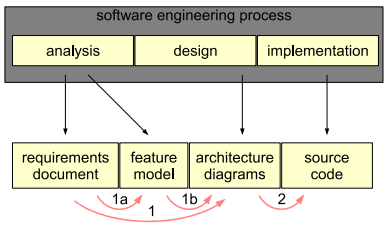

Figure 1: Abstraction Gaps

Different forms of SEP exist: *Waterfall*, *Iterative*, *Extreme Programming* (XP) and *Agile Programming*. But every project, con-

sciously 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 [Hel04] 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 1) 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.

1.2 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 [HKBP03] treat communication between a computer system and a human being differently than that *among* computer systems.

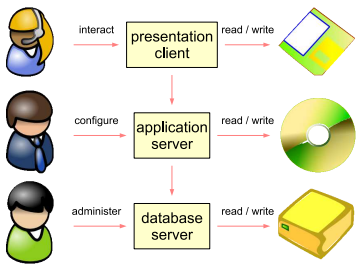

Figure 2: Universal Communication

Figure 2 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*.

1.3 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?

The model of a *Horse* shall serve as example to investigate this further. Figure 3 shows

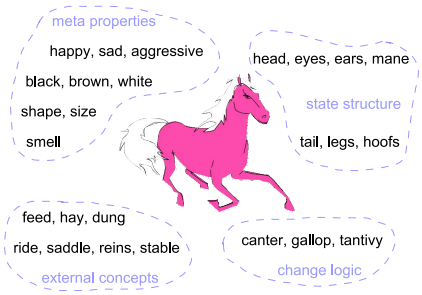

Figure 3: Concept of a Horse

a number of terms commonly used to describe 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 [Hel04]), 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 some-

thing. 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 distinguishes structural- from meta information. Actions (like the gallop of a horse) causing change in the model or its environment are called *Logic* in this work, since they follow certain rules.

2 Architectural Troubles

The conceptual mistakes mentioned in section 1 are partly the reason for, and partly they are caused by incomplete programming paradigms. Of the three abstraction principles of human thinking described in [Hel04], OOP implements *Discrimination* and *Categorisation* only. *Composition* as third kind of abstraction leading to hierarchical (tree-like) models, is not considered.

Hierarchies are not new, they are present in many ways in today's programming. There are object hierarchies, process hierarchies, design patterns modelling a hierarchy and more. But: the hierarchy as concept is not *inherent* in the type system of current programming languages. If it were, then *every* type would be a *Container* by default. Section 4 will introduce such a universal type.

Yet what are the results of that incomplete type system? First and foremost, it is the reason for the existence of multiple kinds of container types, and therewith the reason for falsified contents when using container inheritance, as demonstrated in [Nor]. The lack of a general, container-like type leads to many strong dependencies, which could be avoided when holding type attributes as neutral elements. The language and interpreter of this work use just one structure for knowledge representation, that covers many of the traditional forms of containers.

Further, the bundling of attributes and methods in an OOP class forces classes to not only relate to other classes for accessing their attributes, but also for using the methods offered by them. This often leads to unfavourable bidirectional dependencies [HSP05], that many software patterns even use on purpose (which is a mistake, however [HSP05]). A related problem is that, despite multiple relations in a huge class framework, it is often difficult or impossible to reach some instances along normal object associations, which necessitates the introduction of statically (globally) accessible parts, with all disadvantages [HSP05]. The knowledge schema introduced later on allows to build models with unidirectional relations only, that are easy to navigate, without global access.

Other software design solutions like *Concern* interfaces used in *Component Oriented Programming* (COP) [BMF02] or the *Join Point Model* (JPM) known from *Aspect Oriented Programming* (AOP) [Pro02] have their own drawbacks. Concerns spread functionality and cause redundant code through overlapping interfaces [Hel02], which would be avoidable using an ontological architecture [HKBP03]. The JPM contains some unsolved issues, pointed out by [HT04]. Models as proposed in this article are ontologies.

System Family Engineering applies a so-called *Six-Pack* approach [CMU03, EP01], based on the separation of *Domain Engineering* (DE) and *Application Engineering* (AE). The work described in this article proposes a separation of knowledge and system control.

3 Approach

On its way to solving the issues mentioned in sections 1 and 2, the work followed the *Cybernetics Oriented Programming* (CYBOP) approach [Hel04]. The idea behind is as simple as it is helpful; it suggests to:

Inspect solutions of various disciplines
of science, phenomenons of nature,
and apply them to software engineering.

Figure 4 shows some 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 distinction between *Statics and Dynamics*, a new kind of *Knowledge Schema* and a separation of *State- and Logic* models.

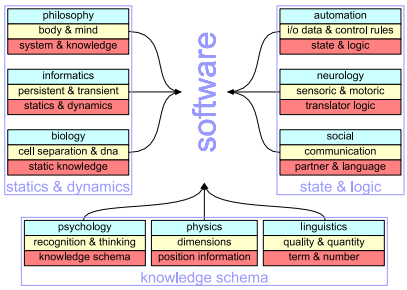

Figure 4: Mindmap of Influential Sciences

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* [Par97, Kue01].

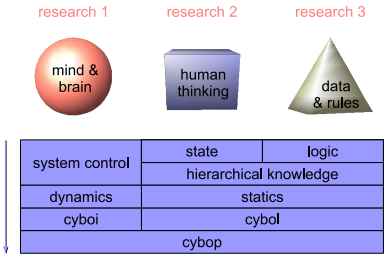

Figure 5: Overall CYBOP Approach

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* [Hel04], as investigated by psychology. Through their application, knowledge becomes *hierarchical*. Moreover, this work tries 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 5), all knowledge (states and logic), belongs to a system's *Statics*, and is described by CYBOL language templates (section 5). The processing of knowledge at runtime, to control a system, is *Dynamics* and happens in the CYBOI interpreter.

4 Inter-Disciplinary Ideas

Many scientific fields (section 3) have been touched and delivered ideas for this work, not all of whom can be mentioned or elaborated in this article. A few examples shall be given, though; one for each proposal.

4.1 Statics and Dynamics

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 [HSP05] *Layers*, *Domain Model* and *Model View Controller* (MVC) all suggest to apply this principle.

The *Tools & Materials* approach [Zea04] talks of *active* applications (tools) working on *passive* domain data (material). And also *System Family Engineering* (section 2) 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 6). 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.

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* [HSP05], 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

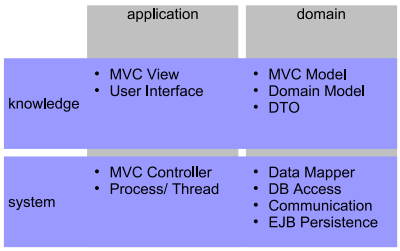

Figure 6: Different Knowledge Separations

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 interdependencies 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) [HSP05] – 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 other data models which are traditionally hold in the application layer, should rather belong to the domain layer, together with all other business knowledge.

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* [HM03] 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 7).

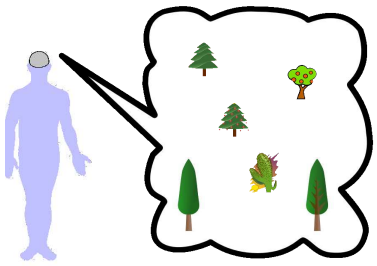

Figure 7: Data Garden

The interpreter 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.

Further arguments in favour of a distinction of statics and dynamics are: mind & body (philosophy), cerebral cortex & communication regions (neurology), genetic information & cell body (biology), long- & short-term memory (psychology), and more.

4.2 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.

4.2.1 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* [Hel04]. Each model *knows* about the parts it consists of.

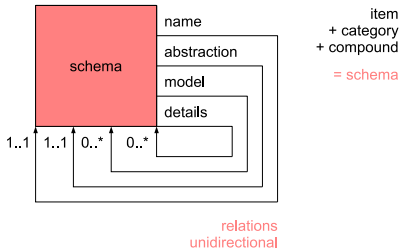

Figure 8: Knowledge Schema

Yet what does this knowledge of a compound model (whole) about its parts imply? Software developers call knowledge *about* something *Meta Information*. Figure 8 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 *Abstraction* (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*.

4.2.2 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 [Hel04] 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.

Figure 9 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

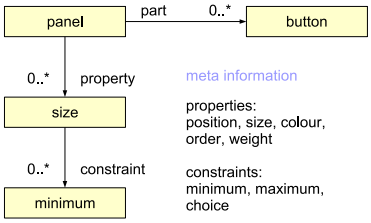

Figure 9: Double Hierarchy (Parts | Meta Info)

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*.

4.3 State and Logic

According to the observations made in the work described in this article, there are two kinds of knowledge: *State*- 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 [Hel02]. 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* [CKP00]. The omnipresence of hierarchies in the MVC was demonstrated in [HBKP03].

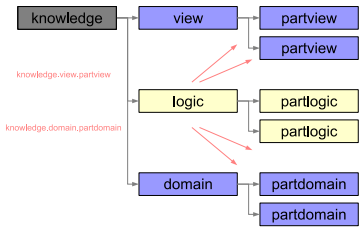

Figure 10: Runtime Logic manipulating States

Figure 10 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 5), and be reloaded the next time an application starts, this may be seen as a kind of *Meta Programming* [Con04].

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 [HKBP03] simplify architectures and unify communication.

5 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* [Pro04] was implemented in CYBOL, but will – due to the limited space – not be explained further here.

5.1 Document Type Definition

Since CYBOL is based on the *Extensible Markup Language* (XML), a *Document Type Definition* (DTD) can be given (figure 11).

```
<!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>
```

Figure 11: CYBOL DTD

One can recognise the purely hierarchical structure as described by the CYBOP knowledge schema (section 4.2.1). The three elements *part*, *property* and *constraint* have the same list of required attributes.

5.2 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>
```

5.3 Container Mapping

State-of-the-art programming languages offer a number of different container types, partly based on each other through inheritance. Section 2 of this work identified *Container Inheritance* as one reason for falsified program results.

Section 4.2 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.

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: Mapping Containers to CYBOL

5.4 Knowledge-handling System

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.

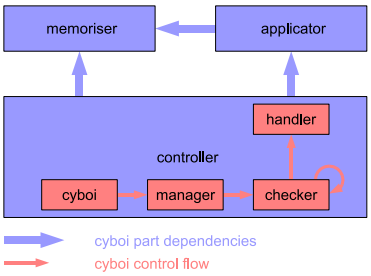

Figure 12: CYBOI Architecture

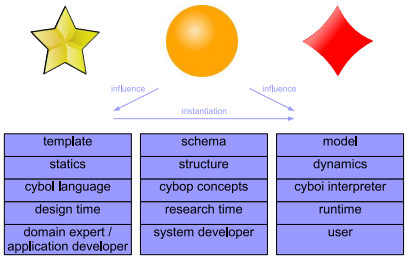

Figure 13: Knowledge Triumvirate

Figure 12 shows three main parts of CYBOI: 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*. Parallels to the *von Neumann* architecture [Tea05] are intended.

6 Summary and Future

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.

The results can be reduced to one illustration: the *Knowledge Triumvirate* (figure 13). 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 portable.

Although this work does not address the *Software Engineering Process* (SEP) directly, its results have great effect on it. Section 1.1 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 1). Since knowledge gets interpreted directly, the formerly needed implementation phase disappears.

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

[BMF02] Federico Barbieri, Stefano Mazzocchi, and Pierpaolo Fumagalli. *Apache Jakarta Avalon Framework*. Apache Project, 2002.

[CKP00] Jason Cai, Ranjit Kapila, and Gaurav Pal. HMVC: The layered pattern for developing strong client tiers. Java World Online Magazine, July 2000.

[CMU03] Software Engineering Institute Carnegie Mellon University. Domain Engineering: A Model-based Approach. Website containing technical Reports, 2003.

[Con04] Collaborating Contributors. Wikipedia – The Free Encyclopedia. Web Encyclopedia, October 2004.

[EP01] ITEA Project 99005 Eureka! 2023 Programme. Engineering Software Architectures, Processes and Platforms for System Families (ESAPS), September 2001.

[HBKP03] 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.

[Hel02] Christian Heller. Cybernetics Oriented Programming (CYBOP) in Res Medicinae. In *OSHCA Conference Online Proceedings*, Los Angeles, November 2002. Open Source Health Care Alliance (OSHCA).

[Hel04] Christian Heller. Cybernetics Oriented Language (CYBOL). *IIIS Proceedings: 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004)*, V:178–185, July 2004.

[HKBP03] Christian Heller, Torsten Kunze, Jens Bohl, and Ilka Philippow. A new Concept for System Communication. *Ontology Workshop at OOPSLA Conference*, October 2003.

[HM03] Herwart (Wau) Holland-Moritz. Der Datengarten. Internet Website, 2003. www.wauland.de/datagarden.html.

[HSP05] Christian Heller, Detlef Streitferdt, and Ilka Philippow. A new Pattern Systematics. http://www.cybop.net, March 2005.

[HT04] Stephan Huttenhuis and Nick Tinnemeier. The Join Point Model (JPM) in Aspect Oriented Programming (AOP). March 2004.

[Kue01] Ralf Kuehnel. *Agentenbasierte Softwareentwicklung: Methode und Anwendungen*. Agenten Technologie. Addison-Wesley, Muenchen, 2001.

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

[Par97] David Parks. Agent Oriented Programming: A Practical Evaluation. Web Article, May 1997.

[Pro02] AspectJ Project. AspectJ: Aspect-Oriented Java Extension, 2002. http://aspectj.org.

[Pro04] Res Medicinae Project. Res Medicinae – Medical Information System, 1999-2004. http://www.resmedicinae.org.

[Tea05] SelfLinux Team. *SelfLinux – Linux-Hypertext-Tutorial*. PingoS e.V., Hamburg, 0.11.3 edition, June 2005.

[Zea04] Heinz Zuellighoven and et al. Tools & Materials Approach to Software-Development. JWAM Open Source Project, 2004.