Hierarchy. To the latter do also belong constraints like the minimum size of a button or a possible choice of colours for it. Constraints can be treated like meta information about properties. Once again: *Properties* are information about a *Part*; *Constraints* are information about a *Property*.

7.3.4 Modelling Example

Another example shall be given to substantiate the need to distinguish between the several kinds of information. How would one describe a *Horse*, unbiassed as a child, by doing some brainstorming? Figure 7.19 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.

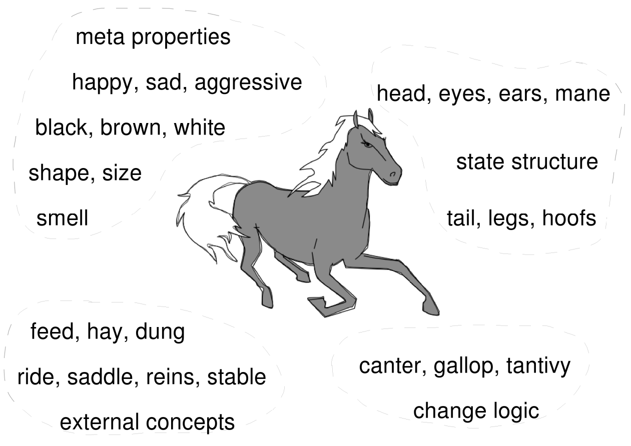

Figure 7.19: Concept of a Horse with Structure, Meta Properties and Logic

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. *Structured- and Procedural Programming* (SPP) (section 4.1.6), for example, would probably use a structure called *struct* or *record* representing the leg and a field standing for the leg's colour. Similarly, *Object Oriented Programming* (OOP) (section 4.1.15) would use a class representing the leg and an attribute standing for the leg's colour, which, in Java source code, would look as follows:

```
public class Leg {
    private String knee;
    private String hoof;
    private String colour;
}
```

However, when following the modelling principles of human thinking (section 7.1), 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.*

It seems that scientists (including the author of this work) and adults in general have unlearnt to think simple like children. Scientists sometimes tend to unnecessarily complicate things that can be described quite easy. Other times, they simplify things which better be distinguished. And looking back into the history of programming, one wonders who ever had this idea of mixing structural elements, properties with meta information and logic algorithms into just one structural entity as at least SPP (record, struct) and OOP (class) do.

The CYBOP knowledge schema introduced before takes care of these things and distinguishes whole-part- from meta information. Actions (like the gallop of a horse) causing some change in the model (horse) or its environment are called *Logic* in this work, since they follow certain rules. Chapter 8 will deal with these.

7.3.5 Container Unification

Section 4.1.15 demonstrated how container inheritance, due to polymorphism, may cause unpredictable behaviour leading to *falsified* container contents. The sections of this chapter 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 that were introduced in section 4.1.15 are *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 (section 4.1.12).

Chapter 9 will introduce a language capable of expressing all aspects of the knowledge schema as proposed in this chapter.

7.3.6 Universal Memory Structure

To better explain the differences between traditional- and cybernetics-oriented design models, a further example shall be given. Figure 7.20 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.

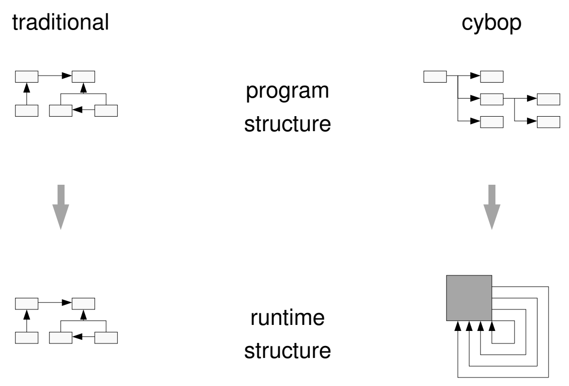

Figure 7.20: Universal Memory Structure

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

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.