On the Cognitive Foundations of Modularity

Miguel P. Monteiro
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, PORTUGAL
mmonteiro@di.fct.unl.pt

Keywords: POP.V.A concepts and categorization; POP.IV.A modularity; POP-II.B. program comprehension; design; coding; maintenance. POP-III.C. procedural/object oriented; POP-IV.A. object oriented design; POP-V.B. literature review.

Abstract

Modularity is a fundamental concept in software engineering with deep roots in human cognition. However, to date few studies of the cognitive roots of modularity have been carried out. To contribute to filling this gap, we examine memory, abstraction and conceptual categorization as viewed in cognitive psychology. We clarify the connections between those topics and hierarchical decomposition. Using them as a basis, we propose a view of software modules as cohesive and structured representations of conceptual categories geared to facilitate reasoning along multiple levels of abstraction. We examine some features of the object-oriented paradigm and point out strengths and limitations in how mainstream object-oriented languages match the cognitive needs of software developers. We conclude by mentioning some future work that may bring insights on the language features to strive for in the next wave of technology adoption.

1. Introduction

Notions of what a software module should be can be traced to the classic paper by Parnas (1972), often cited as the original proponent of the principle of *separation of concerns* (SoC) as the criteria to decompose systems into modules. Modern notions of software module build on the SoC principle and are also strongly influenced by the notion of class from the object-oriented programming (OOP) paradigm. OOP provides strict support for encapsulation, enabling the hiding of implementation details behind well-defined interfaces. Polymorphism reduces the impact of implementation evolution on client modules that depend on the interfaces. OOP is the current "dominant" paradigm and is widely adopted in industry. Some claim that OOP is "natural" on account of going further than previous paradigms in providing a match between modules and concerns from the problem or system under analysis (Booch 1994, Meyer 1997). Not all research yields a clear-cut support for the "naturalness" claim of OOP (Détienne 2001). A study by Sheetz and Tegarden (2001) reaches the surprising conclusion that OOP is simultaneously "natural" and hard to learn.

Despite the aforesaid desirable features, OOP is subject to a number of well-known limitations that suggest that OOP does not provide a perfect match between software modules and concerns as formed by the human mind. One limitation currently driving many research efforts is the *tyranny of the dominant decomposition* (Tarr et al. 1999), which expresses that all paradigms, OOP included, support a *single*, primary decomposition criterion for a software system. As a consequence, concerns that do not align with the primary decomposition tend to cut across the decomposition units, i.e., OOP fails to modularize non-aligning or *crosscutting concerns* (Kiczales et al. 1997). In OOP, non-aligning concerns tend to be non-functional since OOP is good at modularizing functional concerns. An important difference between OOP and older paradigms is that the particular decomposition OOP advocates – into entities with state plus behaviour over that state – seems to be more advantageous (Sheetz and Tegarden 2001). However, the root cause of the crosscutting problem is that *any* hierarchical decomposition faces difficulties when dealing with problems whose structure of concerns is multidimensional (Kiczales et al. 1997, Tarr et al. 1999, Czarnecki 1998).

This research starts from the assumption that software modules should ideally be representations in software of the concerns, or *concepts*, formed by the human mind when dealing with a given problem

or task. It is motivated by the desire to answer the following questions: what characteristics of human cognition cause the tendency to organize concepts hierarchically? In what ways does the human mind deal with structures of concepts that cannot be adequately represented hierarchically? To provide some answers, the paper comprises the following sections. Section 2 presents a survey of topics drawn from cognitive psychology that are relevant to modularity. Next, section 3 discusses some features of programming language technology in light of a desirable match between software structures and structures of concepts. Section 4 points out opportunities for future work and concludes the paper.

2. Relevant Notions of Cognition

This section provides a short review of memory, abstraction and categorization as viewed in cognitive psychology.

2.1. Human Memory

Human memory comprises a system in which several main components can be discerned. For the purposes of this survey, the two important components are *long-term memory* (LTM), which is of virtually unlimited in capacity and duration, and *working memory*, which can store just a few of units of knowledge for periods of just a few seconds. From LTM one can recall general information about the world, learned on previous occasions, past experiences, specific rules previously learned, etc. Working memory is the sole component whose contents are immediately available to thought processes. Often, the information elements fed to working memory originate from LTM, but can also be fed directly from external stimuli, i.e., from the various sensory registers associated to the senses.

Two fundamental features of working memory dictate important limitations of thought processes. First, the storage of working memory comprises a limited number of "slots" capable of holding information elements. Second, though the slots are fixed in number, the information content (often called *chunk*) in each slot is *variable*, ranging between trivial items, such a single digit or letter, to high-content items such as complex concepts. Miller (1952) reported on experiments suggesting that the capacity of working memory is 7±2 items, hence the title of his famous paper, "The magical number, plus or minus two". More recent research suggests that 7±2 is an overestimate, probably being the by-product of additional mental processes such as the use of chunking, the technique of recoding individual items into groups. Chunking yields hierarchical representations in memory and forms the basis for mnemonics. A more reasonable estimate of the number of slots, obtained when chunking and mnemonics are prevented, is around 4 (Cowan, 2001).

Mnemonics are about anchoring a new information item we want to encode on an item already stored in LTM. Many different techniques for mnemonics exist (Baddeley et al. 2009, Groeger 1997). What is worth noting here is that any technique directly depends on what knowledge a person already possesses – in her LTM. To illustrate chunking and mnemonics, consider the task of memorizing the sequence 20010911. Memorizing each of the 9 digits separately is beyond the capacity of many people's working memory. A straightforward use of chunking would entail regrouping the digits into three-digit chunks, e.g., 200 109 11, which is a more effective way to memorize numbers – taking three "slots". This organization is often employed to represent, e.g., telephone and social security numbers. However, we may notice a relation between the sequence and the date of the terrorist attacks (as in, e.g., 2001-09-11), in which case we can take advantage of it by encoding the sequence as the concept "September Eleven", which takes just one "slot". However, this particular instance of mnemonics could not have been used before 11th September 2001, when a special meaning was not attached to that date – and held in LTM.

It is worth pointing out that chunking and mnemonics are a basic property of human memory, not just techniques used only after being learned – though some are certainly amenable to improvement through training (Ericsson et al. 1980). For example, the experiments conducted by Feigenson and Halberda (2008) show that even untrained preverbal infants have the capacity to organize conceptual knowledge hierarchically in order to enhance memory capacity.

2.2. Concepts and Categorization

Concepts are mental representations of categories of objects, ideas, or events that have a common set of features (Czarnecki 1998, Murphy 2002). To recognize an object as an instance of a concept is usually referred to as *classification* or *categorization* (Rosch 1973, Rosch et al. 1976, Murphy 2002). A *concept* is the mental representation of a class of things and a *category* is the class itself (Murphy 2002). In everyday tasks, we must rely on our concepts of the world to make sense of it. In doing this, two extremes must be avoided: (1) to perceive every new object or entity we encounter as a new concept, in which case concepts would be useless as we would be overwhelmed by an excess of information; and (2) to always perceive each new object as an instance of the same concept, which would miss the whole point of concepts. The human mind avoids both extremes by using a set of concepts whose number is kept manageable but whose members are informative enough to be useful (Murphy 2002). Thus, we can recognize a green apple and a brown apple as instances of the same concept and still distinguish a yellow apple from a yellow pear. Rosch et al. use the term *cognitive economy* to refer to this ability to reduce the potentially infinite differences among stimuli (objects) to behaviourally and cognitively usable proportions (Rosch et al. 1976). The importance of *cluster analysis* in statistics is due to its capability to mimic similar capabilities of the human mind.

The connection between chunks as contents of working memory and concepts is that in many cases, the chunks *are* concepts – or at least cues for concepts and their constituent parts, from which the parts can be quickly retrieved to feed thought processes. We can make a rough analogy with a data structure that may not hold large data sets but is able to hold pointers from which the data sets can be quickly retrieved.

A concept can be an item dense with information, yielding an efficient way to use limited cognitive resources. For instance, the concept of *Dog* has an enormous amount of information associated to it (e.g., has fur, is a mammal, has four legs, barks, etc.), but we have the ability to retrieve from the LTM only those items that are needed for a given task, which suggests that the internal representation of concepts is hierarchical.

```
public void add(Object element) {
   if(!readOnly) {
      int newSize = size + 1;
   if(newSize > elements.length) {
      Object[] newElements =
         new Object[elemnts.length + 10];
      for(int i=0; i<size; i++)
         newElements[i] = elements[i];
         elements = newElements;
      }
      elements[size++] = element;
   }
}
```

→

```
public void add(Object element) {
   if(readOnly)
      return;
   if(atCapacity())
      grow();
   addElement(element);
}
```

Figure 1 – Compose Method refactoring (Kerievsky 2004).

Rosch (1976) points out that a particularly important property of a concept, probably the most often used, is the concept's *name*[1]. To illustrate its importance, consider the example used by Kerievsky (2004) to illustrate the *Compose Method* refactoring (Fowler 1999). Kerievsky (2004) characterizes the action of *Compose Method* by saying "Transform the logic into a small number of intention-revealing steps at the same level of detail". Figure 1 shows two versions of a method, before and after applying *Compose Method*. The first version is not very complex, but it is complex enough that one needs to think about how it carries out its task. The second version shows what can be achieved by applying chunking to source code. The refactoring entails applying *Extract Method* (Fowler 1999) a number of times, which is always an opportunity to raise the level of abstraction because it enables us to add *names* that provide a gain in documentation. There is a gain because method names represent concepts and the level of abstraction in the source code is thus raised. The example also illustrates Fowler's dictum (1999) that "If you have a good name for a method you don't need to look at the

[1] This fact points to strong linguistic connections: humans use language to communicate concepts to other humans. However, this paper is focused on the cognitive, not linguistic, connections. Therefore linguistics connections are out of scope of this paper.

body." Naturally, a name is "good" when it represents the underlying concept precisely and clearly. Blackwell et al (2008) make a similar point albeit from a different perspective. These considerations also explains why using single letters as the names of variables or methods is generally not a good idea, with the possible exception of letters to which we attach a special meaning in a context, such as the letter "i" often used as counters within for loops. The meaning of "i" in that context is already stored in the LTM of most programmers.

2.3. Abstraction

Abstraction can be a surprisingly multi-faceted notion. This section summarizes views of abstraction from software engineering and from the cognitive sciences.

Definitions of abstraction in software engineering exist in many variants, always stressing the ability to remove irrelevant details so that one can focus on the relevant details that remain. One definition by Czarnecki (1998) reads: "abstraction involves the extraction of properties of an object according to some focus: only those properties are selected which are relevant with respect to the focus (e.g. a certain class of problems). Thus, abstraction is an information filtration process which reduces the initial amount of information to be used in problem solving". See also Greenfield et al. 2004 and Nicholson et al. 2009. The inverse operation to abstraction is refinement or concretization. Greenfield et al. (2004) define *refinement* as making a description more complex by adding information. Refinement is used in software development to produce executables from requirements, which are seen as the starting point for software development. Developers start with requirements and progressively produce more concrete descriptions of the software, such as analyses, designs, implementations and ultimately executables, by adding information. Greenfield et al. (2004) also note that adding information usually entails *design decisions*, i.e., choice among alternative solutions.

Gray and Tall (2007) emphasize a different facet of abstraction: *information compression*. They first note that the term *abstraction* has its origins in the Latin *ab* (from) *trahere* (to drag) as:

- a verb: to abstract, (a process),
- an adjective: to be abstract, (a property),
- and a noun: an abstract, for instance, an image in painting (a concept).

Gray and Tall note that the corresponding term *abstraction* is dually a process of 'drawing from' a situation and also the concept (the abstraction) output by that process. It has a multi-modal meaning as process, property or concept. They also note that concepts are noticed before they are named. First, various properties and connections are perceived in a given phenomenon, but it is only when these are verbalized and the phenomenon is *named* that humans begin to acquire power over it, namely to talk about it and refine its meaning in a more serious analytic way. These remarks point to the strong connections between concepts and linguistics.

The notion of abstraction expressed by Gray and Tall can be summarized as the ability to compress the information contained in a complicated subject by reducing it to a single representative entity. In most cases, that representative entity is a concept. The ability to compress information is important because cognitive resources are limited. Gray and Tall propose that it is the underlying mechanism of abstraction to compress phenomena into concepts that enables human thought in general and mathematical thinking in particular to operate at successively higher levels of sophistication.

Past PPIG workshops covered the topic of abstraction. Blackwell et al. (2008) call into attention the dangers of the use of abstraction in the context of software development and evolution tasks and provide concrete examples. Nicholson et al. (2009) characterize the cases described by Blackwell et al. as failures to find the right abstractions and provides guidelines on the proper use of abstraction in the context of software development and evolution tasks, using the examples reported by Blackwell et al. However, those works do not analyse abstraction as a mechanism of the mind – the focus of this paper – and their purpose is not to shed light on what structures of concepts are suitable for the use of abstraction (e.g., whether they should be hierarchical). That is the topic of the following section.

2.4. Abstraction and Hierarchical Decomposition

Hierarchical decomposition is pervasive not just in programming paradigms but in most conceptual creations of Mankind. Examples abound in fields as diverse as classical physics, in which physical objects are hierarchically decomposed into molecules, atoms, and subatomic particles; to biological classification, which divides living organisms into a hierarchy of domain, kingdom, phylum, class, order, etc. In software engineering, a programming paradigm is defined by the criterion used for decomposition. Though Parnas is often cited in relation to modularity (1972), it is perhaps less often noticed that his classic paper also calls into attention to the importance of hierarchical decomposition, a subject to which Parnas returned in subsequent research. Parnas et al. (1984) advocate hierarchical decomposition as a way to tackle the problem of managing a large number of modules that make up a big system.

One key property of abstraction is that for abstraction to work, the structure of concepts on which it operates must be organized into hierarchies or taxonomies. In this context, hierarchies are understood as sequences of progressively broader categories, in which each category includes all the previous ones. In other words, it is not enough to obtain some *tree* structure: the relations between the nodes of the tree must have the *set inclusion* relation, also known as the *is-a* relation (cf. Murphy 2002, chapter 7). This structural characteristic is observed in the class inheritances of OOP with respect to objects. For instance, if a class hierarchy includes a super-class *Employee* and a sub-class *Manager*, code that manipulates instances of *Employee* is equally valid for instances of *Manager*.

Whenever thought processes apply abstraction, some set-subset hierarchical relationship is reified, comprising at least two levels: that of the set of elements comprising the subject and that of the concept representing the set. The process of information compression can be applied multiple times, yielding multi-level categorizations, such as those represented in deep-nested class diagrams. Another reason why set-subset hierarchical relationships are a prerequisite for abstraction is *asymmetry*: different levels of the representational hierarchy support different inferences. An inference about a *mammal* (e.g., have hair, three middle ear bones and mammary glands functional in mothers with young) does not necessarily apply at the more abstract *animal* level. However, an inference about a mammal applies to all levels nested within it (e.g., *dog* and *algarvian water dog*). Finally, set-subset hierarchical relationships possess the *transitivity* property: we generally have thoughts such as "all pines are evergreens and all evergreens are trees; therefore, all pines are trees". Transitivity enables *property ascription*, e.g. when learn of new instance of a category, we can assume it has the properties generally ascribed to that category.

In cognitive psychology, it is still an open issue whether hierarchical representations are directly encoded in LTM or are generated "on-the-fly" from some other representation – and note that these hypotheses are not mutually exclusive. However, it is consensual that hierarchical representations play an important role in enabling the efficient use of limited cognitive resources.

If we fail to reify a hierarchical structure from a given subject (i.e., some elements are left "out"), abilities such as inference making and property ascription can no longer be safely made. If we experience difficulties in reifying such a hierarchy when dealing with a complex problem, we usually complain that the "right abstractions" were not yet found (cf. Nicholson et al. 2009, Blackwell et al. 2008).

2.5. On the Fuzziness of Concepts

One facet of concepts and categories deserving a specific discussion is their *fuzziness at the boundaries*, i.e., things are members of a category *to a degree* (Rosch 1973, Rosch and Mervis 1975). For instance, not all red things are equally red – some are redder than others.

The original, *classical view* of concepts postulated that concepts are mentally represented as definitions. Every object is either in or not in the category, with no in-between cases. This view is discredited today but held in some form or another for a surprisingly long time – from the time of Aristotle (cf. Apostle 1980) to the work of Rosch in the 1970s (1973, 1975, 1976) – and there was

considerable resistance from some circles to its overturning[2]. A corollary of the classical view is that the concepts we form when approaching the external world always follow a hierarchical structure. However, people disagree on whether particular objects are members of a given category. To illustrate, we mention some results from an experiment in which subjects were asked to make a number of category judgments (McCloskey and Glucksberg 1978). Thirty subjects agreed that *cancer* is a disease and *happiness* is not, but sixteen thought that *stroke* is a disease while the rest didn't. Disagreement over category boundaries occurs not only *among* subjects, but also *within* subjects, depending on context and circumstances. For instance, the same thirty subjects repeated the test at about one month later and eleven of them reversed themselves on *stroke*. Reversals occur often because categorizations depend on context and purpose, which can change often and rapidly.

The existence of gradations in deciding whether a given object is a member of a given category – with some members being judged "central" and others more "marginal" or "atypical" – are referred as *typicality effects* (Murphy 2002) and are pervasive in tasks of categorization. People change the categorizations of atypical members often but are generally stable regarding the typical. The only significant set of concepts free of typicality effects and fuzziness at the boundaries are *mathematical concepts*, i.e., concepts that have a precise definition comprising necessary and sufficient properties. A good example is *square*, which is can be precisely defined as a closed figure that has four sides equal in length and equal angles (cf. chapter 2 of Czarnecki 1998). All other kinds of concepts are referred as *natural concepts*[3].

The multi-faceted nature of concepts deserves a note. Objects are members of many categories simultaneously (Murphy 2002). For example, people fit into many different categories such as being a woman, a reporter, a political conservative, a New Yorker, an African-American, a cousin, etc. Since the world consists of shades and gradations of seemingly infinite variety, resorting to a finite number of concepts is bound to betray symptoms of fuzziness. In addition to directly contradicting the classical view of concepts, an important implication is that *the world is naturally multi-dimensional*. This helps to account for the phenomena of multi-dimensionality in software that give rise to crosscutting and the tyranny of the dominant decomposition.

Interestingly, people do not seem to be troubled by definitional problems when using concepts in informal communication and everyday life. When we use the word *furniture* in a phrase, we do not usually stop to ponder whether curtains and pianos are included (unless we have a specific reason for considering curtains and pianos). In fact, one lesson from categorization in cognitive psychology is that fuzziness is a prerequisite for cognitive economy (in the sense used by Rosch at al. 1976) and therefore an essential property in dealing effectively with the external world.

Presently, cognitive psychology is testing several competing theories to account for the way humans acquire, represent, combine and process concepts (Murphy 2002, Margolis and Laurence 1999). However, none is able to account for the full panoply of observed phenomena. As acknowledged by some senior researchers, the field is struggling in finding a unified theory (Murphy 2002). One reason is that the human mind seems to use different pathways and/or constructions depending on the circumstances. In contrast, the various theories of concepts are focused on particular ways (at least that is how they were formulated initially). When an experiment contradicts a given theory, it is hard to know whether it the theory is wrong or the mind uses a different way for the particular circumstances of the experiment. However, the theories have been serving as vehicles to drive assessments whose results add to the body of knowledge on human cognition. This short survey is focused on that body of knowledge rather than on the theories themselves.

[2] In his book, Murphy (2002) describes attempts from some circles, namely philosophers (cf. chapter 2 of Apostle 1980), to revive the classical view. Murphy considers these attempts to be unconvincing and notes that much support for the classical view after Rosch really amounts to criticism of the evidence against it rather than providing compelling account for observed data. Murphy muses on the possible reasons for this attitude: "(..) there is a beauty and simplicity in the classical view that succeeding theories do not have. (..) To be able to identify concepts through definitions of sufficient and necessary properties is an elegant way of categorizing the world (..)."

[3] Czarnecki (1998) considers other categories of concepts, such as object concepts (e.g. *dog*, *table*), abstract concepts (e.g. *love*, *brilliance*), scientific concepts (e.g. *gravity*, *electromagnetic waves*). In this paper, we find it sufficient to consider just two main categories – those that have a precise definition – mathematical concepts – and all the others, which we term *natural concepts.*

The implication of fuzziness in natural concepts is that there is a significant gap between *internal* representations in the mind and *external* representations of any form. In his famous paper "No Silver Bullet", Brooks (1986) muses on the reason why the pressure for change in software is significantly greater than for computers or automobiles. Brooks writes that software "is pure thought-stuff, infinitely malleable". However, this judgement is not strictly correct, for it applies to thoughts as they are processed *internally*, but not *external* representations of thoughts – as is the case of any software artefact. Such artefacts, like those from most fields of engineering and beyond, should be precise, mathematical and unambiguous and (in the context of software) amenable to automatic processing.

Czarnecki (1998, section 2.4) notes that OOP class modules correspond to the concepts of the classical view. The major difference is that OOP makes more specific assumptions about objects. They have *state* and *behaviour* and collaborate through *interactions*. Czarnecki also notes that an important consequence of fuzziness is that adequate implementations of concepts have to cover enormous amounts of variability. Each "variation" is likely to give rise to different refinements, associated to different data structures, implementation code, or even module structures.

It is worth noting that externalization of concepts is a very broad but extremely important topic. In many fields, making a specification entails producing precise representations of domain concepts, most of which are natural concepts and therefore fuzzy. This is likely to be always a cause of problems, in virtually all fields of endeavour. To illustrate, we mention just a few fairly well-known examples from very different areas:

- **Biology.** How to classify the platypus in the animal kingdom? (wikipedia: platypus)?
- **Law.** How to define "obscenity" and "child pornography" in law, to establish what is protected by the first amendment of the constitution of the EUA and what is not? (Cohen 2003)?
- **International affairs.** How to define "cyber-attack" for the purposes of a NATO response? (The Economist 2008)
- **Astronomy.** Is Pluto a planet? (wikipedia: planet)

We conjecture that definitional problems in dealing with natural concepts are one of the reasons why software is generally felt to be not soft at all, i.e., brittle and fragile. As in most fields, it is hard for software to account for continuums and gradations.

3. Matching Modules with Concepts

Many definitions of modularity exist. One of most developed is by Baldwin and Clarke (1999, p.63), which structures the concept along two subsidiary ideas that are subsumed by the general concept: (1) coupling and cohesion and (2) abstraction, information hiding and interface:

1. "A module is a unit whose structural elements are powerfully connected among themselves (i.e., high cohesion) and relatively weakly connected to elements in other units (i.e., low coupling). Clearly there are degrees of connection, thus there are gradations of modularity".
2. "A complex system can be managed by dividing it up into smaller pieces and looking at each one separately. When the complexity of one of the elements crosses a certain threshold, that complexity can be isolated by defining a separate abstraction that has a simple interface. The abstraction hides the complexity of the element; the interface indicates how the element interacts with the larger system".

The above definitions are broad and go beyond computer science. Nevertheless they mirror important properties of concepts. This should not be surprising, as it is a direct consequence of software modules being examples of their external representations. Modules "inherit" some of their properties. To illustrate the parallel between cohesion[4] plus coupling and observed psychological behaviour, see Figure 2. When making categorizations, people tend to group things that bear similarities according to specific criteria. These are a manifestation of the use of cohesion and coupling in categorization

[4] Blackwell et al. (2008) use the term "aggregation" to refer to cohesion.

criteria. Note that the criteria used to determine what coheres can vary widely. It can be some superficial property (e.g., physical similarity) but as knowledge is developed it can be a property or set of properties that rely on more sophisticated knowledge (e.g., personality traits, mathematical properties). Thus, a child may initially categorize a whale as a fish because a whale resembles a fish. After learning what mammals are, she may re-categorize whales as mammals.

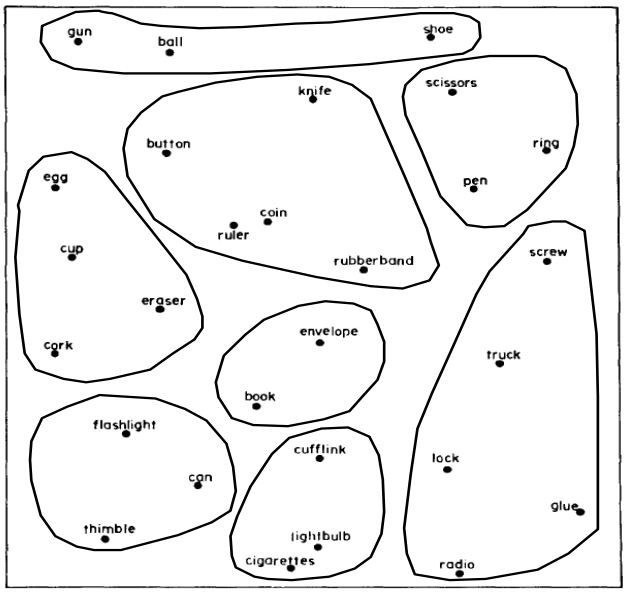

Figure 2. Layout used in a memory experiment in which McNamara et al. (1989) tested subjects for their recalls of 28 objects. Subjects saw these objects laid out in a 20 × 22 ft area (actual objects appeared in layouts, not names of objects). After analysing their patterns of recall, McNamara et al. found evidence that subjects organized the objects into groups and tended to recall groups of objects always together. Shown are the groups used by one particular subject. There are different ways to group (or chunk) the objects and different people used different groupings, but McNamara et al. were able to infer hierarchical groupings from the recall patterns of individual subjects. The closed lines indicate the grouping used by one subject from the experiment on the set of objects.

4. Examining the Match between Modern Modules and Concepts

This section summarizes some structural properties that software modules should desirably possess to provide a good match between structures of concepts and structures of software modules.

4.1. So, what should a Software Module be?

The primary *reason for being* of modules is to provide representations of concepts in software. Note not all concepts are equally interesting: almost anything we can think of is a concept, e.g., *Childhood*, *Happiness*, *Summer*, *Engine*, *Theatre*, *Dog*, *Four*, OOP, *Yesterday*. Clearly, not all concepts lend themselves to be represented by software modules – and neither do we need them to. In software, we are mostly interested in two sub-categories of concepts: *domain* concepts and *implementation* concepts. Only when some concept is associated to state and/or behaviour for some reason, do we reify it as a software module. *Happiness* is not likely to receive many representations in software. However, if we are dealing with, e.g., a problem involving a happiness factor that is measured through numeric values, representing that particular facet of *Happiness* as a class module begins to make sense.

Both the internal structure of modules and the connections between them should be geared for human cognition. The internals of modules should be cohesive, as that is also a trait of what they represent. Modules should be structured so as to facilitate programmers in their tasks of reasoning with them. As reviewed in section 2.4, limitations in cognitive resources lead thought processes in general to reify set-subset relationships from conceptual knowledge so as to process it in an efficient way. Similarly, software developers need the structure of external representations in software to mirror such structures. For instance, organizing a module's internal structure hierarchically makes it easier to be analysed at multiple levels of abstraction. This helps to explain why the internals of modules in OOP are structured hierarchically – the internals of objects comprise other objects. Hierarchical structures facilitate approaching them at several levels of abstraction, with the *name* of module being the highest level, which maximally compresses all associated information. The *name* of the module is routinely used as its representative in, e.g., conversations between human developers. When more details are needed, developers should be able to unpack them gradually. Popular examples include the following (in ascending order of detail):

1. Simple class diagrams with just the class names plus an outline of their relationships.
2. More detailed class diagrams showing details such as operation signatures and attributes.
3. Javadoc artefacts with sections of text describing items of information such as the purpose of symbolic constants, inner classes and methods, purpose and rules of formal parameters in methods, visible fields and potentially thrown exceptions.
4. The source code itself.

Development environments such as Eclipse[5] are developing support for such gradual unpacking by providing multiple, differently detailed views, e.g., *structure views* and *folding capabilities* that expand and collapse sections of source code such as comment sections and method blocks.

Structures of conceptually related modules should mirror taxonomies of concepts. It is the case of class inheritance hierarchies: instances of super-classes are substitutable by instances of sub-classes. Thus, they represent set-subset relationships. However, class hierarchies are marred by an important limitation, which we describe in section 4.3 when analysing polymorphism.

4.2. On the Relevance of Polymorphism

In this section, we call into attention to a number of cognitively friendly traits of OOP. We argue that its popularity can at least partly be explained in terms of that friendliness.

Information hiding provides an essential support for abstraction: when deciding what to hide inside a module, we decide what we abstract way. But a module also must provide a *front* with which we interact with it, i.e., its interface. The interface also serves to clearly define a boundary between what is hidden and what is exposed. Unfortunately, interfaces suffer from the problems of externalization of concepts: they confine fuzzy concepts within brittle and fragile boundaries. Fuzziness in turn gives rise to variability, which results in many different implementations – or refinements – each of which is suitable for a different context. In turn, the existence of multiple refinements for a given abstraction makes it desirable that switching from one implementation to another should impact neither the interface nor client modules – source code using the abstractions should be the same regardless of the specific refinement used, i.e., bindings between interface and implementations should be *seamless*. Polymorphism and dynamic (late) binding achieve this seamlessness.

Polymorphism can be succinctly defined as the ability of an object to belong to more than one type. It is the way to "navigate" across class inheritances in a way that supports abstraction. Anecdotal evidence and numerous publications suggest that polymorphism is popular among developers of object-oriented systems[6]. Many such systems comprise large and complex module structures that harbour much variability and often must manage multiple refinements of a given abstract declaration. For instance, in the "after" version of method add from Figure 1 the names atCapacity, grow, addElement can be associated to an unbounded number of different refinements. Such names as referred in add reside at a given level of abstraction and their specific implementations reside at a lower level. In complex systems, it is a major benefit to developers that the connection between references to a name and refinements be made seamlessly, i.e., without requiring different code to bind the names to different refinements. This is achieved through dynamic binding, which associates references to names to concrete implementations in a well-understood and predictable way, thus minimizing complexity and favouring stability. These features enable developers to look at "clean" representations of a given concept, free of connection code. This in turn enables them to abstract from the details of the specific implementations when they wish to. Polymorphism and dynamic binding are of course well-known features. The purpose here is to stress their role as enablers/facilitators of the use abstraction on the part of developers when they navigate and reason with large and complex module structures. When analysing source code, developers see many names (Liblit et al. 2006), e.g., of variables, modules and operations. Polymorphism turns some of these names into *variation points* while keeping them "clean". Another way of saying this is that by providing this seamlessness,

[5] Eclipse home page. www.eclipse.org/

[6] For instance, polymorphism takes a key role in the catalogues by Gamma et al. (1995), Fowler (1999) and Kerievsky (2004).

polymorphism contributes to make interfaces a bit less brittle and software more malleable – with lower viscosity and easier to reuse and to reason with.

4.3. On the Limitations of Traditional Polymorphism

We identify two limitations in the support for polymorphism in mainstream languages. First, it supports too few variation points – mostly method calls. References to the name of a class, at the points where instances are created, are not dynamically bound (i.e., constructors are not polymorphic). In consequence, bindings between class names and their definitions are not seamless, which increases complexity and hampers stability. The popularity of design patterns geared to overcome this specific limitation (e.g., *Factory Method* and *Abstract Factory* (Gamma et al. 1995)) attests to its significant impact. Second, traditional polymorphism works only along a *single* dimension – that of the inheritance hierarchy to which the class containing the called method belongs. Method declarations and method definitions must all reside at the same inheritance hierarchy. For instance, polymorphism cannot bind a method call to a block of code residing in a class *outside* the hierarchy. The support for a single hierarchy is the root cause of the tyranny of the dominant decomposition (Tarr et al. 1999). Note that multiple inheritance is a *different* thing. Multiple inheritance does not keep the various hierarchies separate and preserve their integrity. In fact, the usual outcome of multiple inheritance is not a hierarchy at all, rather a directed acyclic graph comprising a (possibly dominant) hierarchy with fragments of additional hierarchies bolted into some of its nodes.

The single dimension of polymorphism highlights an important mismatch between the way thought processes organize conceptual knowledge and traditional modular structures. That a concept corresponds to multiple categorizations has the implication that thought processes are able to integrate *multiple* hierarchies. Note that the criteria deciding the organization of the concept representations directly depend on the *focus of interest* (Nicholson et al. (2009) use the term *aspect of interest*). Change the focus through which a concept is reasoned with and the corresponding decomposition changes accordingly. When reasoning with objects (from the real world or from software), we seem capable of building new categorization hierarchies "on-the-fly" and effortlessly switch the focus – and thus the hierarchy. Each change brings a different hierarchy to the fore, which includes the concept viewed through a different perspective and possibly highlighting different facets. We are also able to relate a member of a hierarchy to corresponding members of other hierarchies (i.e., different views, or facets, of a given concept). An entire hierarchy may correspond to a single concept in a different hierarchy. Mainstream OOP fails to support the integration of multiple, mismatching hierarchies, which is a major cause of the tyranny of the dominant decomposition.

5. Future work and Conclusion

In light of the concepts of human cognition reviewed in this paper, we conclude that OOP provides a good match between modules and concepts as regards internal module structure but has limitations in matching the flexibility with which the mind organizes concepts spanning multiple categorizations. In part, the limitations can be traced to limitations in the way polymorphism is currently supported in mainstream OOP languages. Overcoming those limitations entails adopting more advanced forms of modularity technologies. Unfortunately, authors proposing new modularity technologies (Kiczales et al. 1997, Tarr et al. 1999) do not generally cite the body of knowledge of cognitive sciences in support of their approaches to modularity. Cognitive issues seem to be taken only implicitly. Nevertheless the notion of modularity is ultimately rooted in cognition. When Parnas (1972) proposed a set of guidelines for module decomposition, cognitive psychology was at an incipient state. We believe it is now worth a closer look as a provider of insights on how to further develop modularity technologies, namely to tackle the tyranny of the dominant decomposition. This calls for more developed studies of how concepts are structured, combined and used, to shed light on how to improve the way modules are structured and composed.

The tyranny of the dominant decomposition deserves to be studied from a cognitive perspective, both *per se* and in connection to advanced modularity technologies that promise to tackle it. One such technology is aspect-oriented programming (Kiczales et al. 1997), which eschews polymorphism in

favour of an approach to module composition based of joinpoint capture and code blocks triggered when program execution reaches the specified joinpoints. Different technologies comprise different ways to map abstractions to refinements. The relative advantages and disadvantages in terms of consequences to software developer cognition.

Traditional OOP is amenable to improvements. Some advanced OOP languages support a richer set of variation points, addressing the limitations pointed out in section 4.3. *Virtual classes* (Madsen and Moller-Pedersen 1989) and *family polymorphism* (Ernst 2001) turn class names into variation points, even in expression that instantiates object instances. Several languages with these features were proposed in the latest decade, namely gbeta, Object Teams, CaesarJ and Scala. Do they provide a better match to the cognitive needs of programmers than traditional OOP? One of those languages – Object Teams (Herrmann 2007) – was specifically designed to provide direct language support for the concept of *roles* (e.g., as understood by Steimann (2000)), which are represented as separate class hierarchies and seamlessly composed to traditional classes through a role playing relation. It would be interesting to assess what cognitive benefits of these capabilities add to traditional OOP.

Acknowledgements

We thank Maria do Rosário Duarte for providing us hard-to-obtain publications on cognitive psychology. This work is partially supported by project AMADEUS (POCTI, PTDC/EIA/70271/ 2006) funded by *Fundação para a Ciência e Tecnologia*, Portugal.

References

The Economist (2008) *Marching off to cyberwar*. Article at print edition of December 6th.

Wikipedia: "Definition of planet" (accessed 2011-07-05) en.wikipedia.org/wiki/Definition_of_planet

Wikipedia: "Platypus" (accessed 2011-07-05) en.wikipedia.org/wiki/Platypus

Anderson J. R. (2005) *Cognitive psychology and its implications, 6th edition*. Worth Publishers.

Apostle H. (1980) *Aristotle's Categories and Propositions (De Interpretatione)* Grinnell, IA: Peripatetic Press.

Baddeley A., Eysenck M., Anderson M. (2009) *Memory*. Psychology Press.

Baldwin C., Clarke K. (1999) *Design Rules. Vol. 1, The Power of Modularity*. MIT Press.

Blackwell A., Church L., Green T. The Abstract is 'an Enemy': Alternative Perspectives to Computational Thinking. Psychology of Programming Workshop (PPIG 2009), Lancaster University, UK.

Booch G. (1994) Object-Oriented Analysis and Design with Applications, 2nd Ed. Addison Wesley Longman.

Brooks F. (1986) *No Silver Bullet: Essence and Accidents of Software Engineering*. Proc. of the IFIP Tenth World Computing Conference, 1069-1076.

Bruning R., Schraw G., Ronning R. (1995) *Cognitive Psychology and Instruction, 2nd edition*. Prentice Hall.

Cohen H. (2003) *Obscenity, Child Pornography, and Indecency: Recent Developments and Pending Issues*. CRS Report for Congress, The Library of Congress.

Cowan, N. (2001) *The magical number 4 in short-term memory: A reconsideration of mental storage capacity*. Behavioral and Brain Sciences (24) 87-185.

Czarnecki K. (1998) *Generative Programming – Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Ph.D. Thesis, Technical University of Ilmenau, Germany.

Détienne F. (2001) *Software Design – Cognitive Aspects*. Springer.

Ericsson, K. A., Chase W.G., Faloon S. (1980) *Acquisition of a memory skill*. Science, 208, 1181-1182.

Ernst E. (2001) *Family polymorphism*. Proc. of ECOOP 2001. Springer LNCS vol. 2072, 303-326.

Eysenck M., Ellis A., Hunt E., Johnson-Laird P. (eds.) (1990) *The Blackwell Dictionary of Cognitive Psychology*. Blackwell Publishers.

Feigenson L. Halberda J. (2008) *Conceptual knowledge increases infants' memory capacity*. Proc. of the National Academy of Sciences, vol.105, 9926-9930.

Fowler M. (1999) *Refactoring – Improving the Design of Existing Code*. Addison Wesley Longman.

Gamma, E.; Helm R., Johnson R., Vlissides J. (1995) *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley Longman.

Gray E., Tall D. (2007) *Abstraction as a Natural Process of Mental Compression*. Mathematics Education Research Journal, 19(2), 23-40.

Greenfield J., Short K., Cook S., Kent S. (2004) *Software factories: assembling applications with patterns, models, frameworks and tools*. Wiley.

Groeger J. (1997) *Memory and remembering*. Addison Wesley Longman.

Herrmann S. (2007) *A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java*. Applied Ontology, vol.2 (2), IOS Press, 181-207.

Kerievsky, J. (2004) *Refactoring to Patterns*, Addison-Wesley.

Kiczales G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. & Irwin J. (1997) *Aspect-Oriented Programming*. Proceedings of ECOOP'97, Springer Verlag, LNCS vol. 1241, 220-242.

Liblit B. Begel A. Sweetser E. (2006) *Cognitive Perspectives on the Role of Naming in Computer Programs*. Psychology of Programming Workshop (PPIG 2006), Brighton, UK.

McNamara T., Hardya J., Hirtle S. (1989) *Subjective Hierarchies in Spatial Memory*. Journal of experimental psychology, learning, memory and cognition, vol 15(2), 211-227.

Nicholson K., Good J., Howland K. Concrete Thoughts on Abstraction. Psychology of Programming Workshop (PPIG 2009), University of Limerick, Ireland.

Madsen O. L., Moller-Pedersen B. (1989) *Virtual classes: a powerful mechanism in object-oriented programming*. OOPSLA'89, New Orleans, Louisiana, USA.

Margolis E., Laurence S. (eds.) (1999) *Concepts – Core Readings*. MIT Press.

McCloskey, M. and Glucksberg, S. (1978) *Natural categories:Well defined or fuzzy-sets?* Memory and Cognition, vol.6, 462-472.

Meyer B.(1997) *Object-Oriented Software Construction*, 2nd edition, Prentice Hall.

Miller, G. A. (1956) *The magic number seven, plus or minus two: Some limits of our capacity for processing information*. In Psychological Review, no. 63, 81-97.

Murphy G. (2002) *The Big Book of Concepts*. The MIT press.

Neisser U. (ed). (1987) Concepts and Conceptual Development. Cambridge University Press.

Parnas D. L. (1972) *On the criteria to be used in decomposing systems into modules*. Communications of the ACM 15 (12) 1053-1059.

Parnas D. L., Clements P. C., Weiss D. M. (1984) *The modular structure of complex systems*. Proc. of ICSE '84, IEEE press, 408-417.

Rosch E. (1973) *On the internal structure of perceptual and semantic categories*. In Cognitive development and acquisition of language, Moore (Ed.), Academic Press, 111-144.

Rosch E., Mervis C. B., Gray W., Johnson D., Boyes-Braem P. (1976) *Basic objects in natural categories*. In Cognitive Psychology, no. 3, 382-439.

Rosch E., Mervis C. B. (1975) *Family Resemblances: Studies in the Internal Structure of Categories*. Cognitive Development and the Acquisition of Language, vol.7, 573-605.

Sheetz S., Tegarden D. (2001) Illustrating the cognitive consequences of object-oriented systems development. Journal of Systems and Software 59, Elsevier, 163-179.

Tarr P., Ossher H., Harrison W., Sutton Jr. S. (1999) *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. Proc. of ICSE'99, ACM press, 107-119.