Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams

Daniel Moody and Jos van Hillegersberg

Department of Information Systems & Change Management
University of Twente, Enschede, Netherlands
d.l.moody@utwente.nl

Abstract. UML is a visual language. However surprisingly, there has been very little attention in either research or practice to the visual notations used in UML. Both academic analyses and official revisions to the standard have focused almost exclusively on semantic issues, with little debate about the visual syntax. We believe this is a major oversight and that as a result, UML's visual development is lagging behind its semantic development. The lack of attention to visual aspects is surprising given that the form of visual representations is known to have an equal if not greater effect on understanding and problem solving performance than their content. The UML visual notations were developed in a bottom-up manner, by reusing and synthesising existing notations, with choice of graphical conventions based on expert consensus. We argue that this is an inappropriate basis for making visual representation decisions and they should be based on theory and empirical evidence about cognitive effectiveness. This paper evaluates the visual syntax of UML using a set of evidence-based principles for designing cognitively effective visual notations. The analysis reveals some serious design flaws in the UML visual notations together with practical recommendations for fixing them.

1 Introduction

The Unified Modelling Language (UML) is widely accepted as an industry standard language for modelling software systems. The history of software engineering is characterised by competing concepts, notations and methodologies. UML has provided the software industry with a common language, something which it has never had before. Its development represents a pivotal event in the history of software engineering, which has helped to unify the field and provide a basis for further standardisation.

1.1 UML: A Visual Language

UML is a "visual language for visualising, specifying, constructing and documenting software intensive systems" [25]. The UML *visual vocabulary* (symbol set) is loosely partitioned into 13 diagram types, which define overlapping views on the underlying metamodel (Figure 1). So far, there has been remarkably little debate in research or

D. Gašević, R. Lämmel, and E. Van Wyk (Eds.): SLE 2008, LNCS 5452, pp. 16–34, 2009.
© Springer-Verlag Berlin Heidelberg 2009

practice about the visual notations used in UML (also called *visual syntax* or *concrete syntax*). There have been many academic evaluations of UML, but most have focused on semantic aspects [e.g. 8, 26, 35]. There have also been several revisions to UML, but these have also concentrated on semantic issues, with little discussion about, or modification to, graphical conventions. The lack of attention to visual aspects is surprising given UML's highly visual nature. It is even more surprising in the light of research in diagrammatic reasoning that shows that the *form* of representations has a comparatively greater effect on their effectiveness than their *content* [14, 34]. Apparently minor changes in visual appearance can have dramatic impacts on understanding and problem solving performance [28].

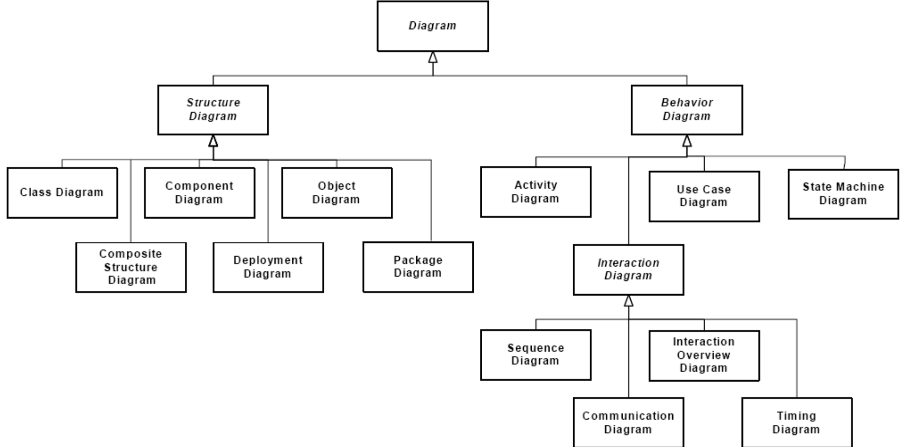

Fig. 1. The 13 diagram types partition the UML visual vocabulary into a set of overlapping sublanguages [25]

1.2 What Makes a Good Visual Language?

Diagrams are uniquely human-oriented representations: they are created *by* humans *for* humans [12]. They have little or no value for communicating with computers, whose visual processing capabilities are primitive at best. To be most effective, visual languages therefore need to be optimised for processing by the human mind. *Cognitive effectiveness* is defined as the speed, ease and accuracy with which information can be extracted from a representation [14]. This provides an operational definition for the "goodness" of a visual language and determines their utility for communication and for design and problem solving.

The cognitive effectiveness of diagrams is one of the most widely accepted assumptions in the software engineering field. However cognitive effectiveness is not an intrinsic property of diagrams but something that must be *designed* into them [14, 28]. Diagrams are not effective simply because they are graphical, and poorly-designed diagrams can be far *less* effective than text [5].

Software engineering is a collaborative process which typically involves technical experts (software developers) and business stakeholders (end users, customers). It is therefore desirable that system representations (especially at the requirements level)

can be understood by both parties: effective user-developer communication is critical for successful software development. However a common criticism of UML is its poor communicability to end users [35]. In practice, UML diagrams are often translated into textual form for verification with users, which is a clear sign of their ineffectiveness for this purpose [36].

1.3 How UML Was Developed

The graphical notations used in UML were developed in a bottom-up manner, by reusing and synthesising existing notations:

> "The UML notation is a melding of graphical syntax from various sources. The UML developers did not invent most of these ideas; rather, they selected and integrated the best ideas from object modelling and computer science practices." [25]

How the "best" ideas were identified is not explained, but seems to have been done (and still is) based on expert consensus. This paper argues that this is *not* a valid way to make graphical representation decisions, especially when the experts involved are not experts in graphic design (which is necessary to understand the implications of choices made). Being an expert in software engineering does *not* qualify someone to design visual representations. In the absence of such expertise, notation designers are forced to rely on common sense and opinion, which is unreliable: the effect of graphic design choices are often counterintuitive and our instincts can lead us horribly astray [40].

Design rationale is the process of explicitly documenting design decisions made and the reasons they were made. This helps provide traceability in the design process and to justify the final design. Design rationale explanations should include the reasons behind design decisions, alternatives considered, and trade offs evaluated [15]. Such explanations are almost totally absent from the design of the UML visual notations. Graphical conventions are typically defined by assertion (e.g. "a class is represented by a rectangle") with no attempt to justify them.

1.4 Objectives of This Paper

We argue that the design of visual notations should be *evidence based*: choice of graphical conventions should be based on the best available research evidence about cognitive effectiveness rather than on common sense or social consensus. There is an enormous amount of research that can be used to design better visual notations (though mostly outside the software engineering field). This paper evaluates the visual syntax of UML 2.0 using a set of evidence-based principles for designing cognitively effective visual notations. Our aim is to be as constructive as possible in doing this: to improve the language rather than just point out its deficiencies. As a result, when we identify problems, we also try to offer practical (though evidence-based) suggestions for fixing them.

2 Related Research

Ontological analysis has become widely accepted as a way of evaluating the semantics of software engineering notations [7, 32]. This involves a two-way mapping between

the constructs of a notation and an ontology: the *interpretation mapping* describes the mapping from the notation to the ontology while the *representation mapping* describes the inverse mapping [7]. Ideally, there should be a one-to-one correspondence between the categories in the ontological theory and notation constructs (Figure 2). If there is not such a correspondence, one or more of the following problems will occur:

- *Construct deficit* exists when there is no semantic construct corresponding to a particular ontological category.
- *Construct overload* exists when the same semantic construct is used to represent multiple ontological categories
- *Construct redundancy* exists when multiple semantic constructs are used to represent a single ontological category
- *Construct excess* exists when a semantic construct does not correspond to any ontological category.

If construct deficit exists, the language is said to be *ontologically incomplete*. If any of the other three anomalies exist, the language is said to be *ontologically unclear*. Ontological analysis has previously been applied to evaluate the semantics of UML [26]. This paper complements this previous research by extending the evaluation of UML to the level of visual syntax (*form* rather than *content*).

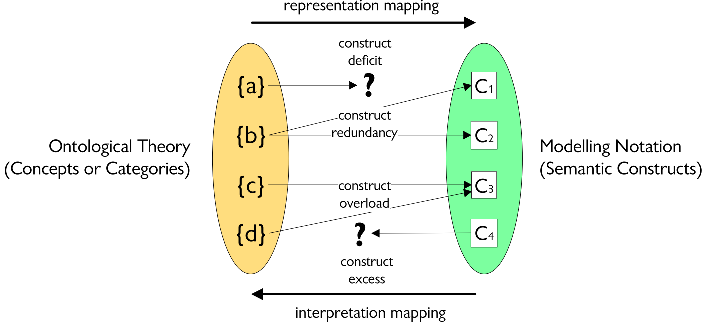

Fig. 2. Ontological Analysis: there should be a 1:1 correspondence between ontological concepts and modelling constructs

3 Principles for Visual Notation Design

A previous paper defined a set of principles for designing cognitively effective visual notations [19]. These principles were synthesised from theory and empirical evidence from a wide range of disciplines and have been previously used to evaluate and improve two other visual languages [20, 21]: *ArchiMate* [13], which has been recently adopted as an international standard for modelling enterprise architectures and *ORES*, a proprietary cognitive mapping method for organisational development and strategic planning. We use these principles in this paper as a basis for evaluating the cognitive effectiveness of the UML visual notations. This section reviews the principles.

3.1 Principle of Semiotic Clarity

This principle represents an extension of ontological analysis to the level of visual syntax using a theory from semiotics. According to Goodman's *theory of symbols* [9], for a notation to satisfy the requirements of a *notational system*, there should be a one-to-one correspondence between symbols and their referent concepts [9]. The requirements of a notational system constrain the allowable expressions and interpretations in a language in order to simplify use and minimise ambiguity: clearly, these are desirable properties for software engineering languages. When there is *not* a one-to-one correspondence between semantic constructs and graphical symbols, the following anomalies can occur (in analogy to the terminology used in ontological analysis) (Figure 3):

- *Symbol redundancy* exists when multiple symbols are used to represent the same semantic construct. Such symbols are called *synographs* (the graphical equivalent of synonyms). Symbol redundancy places a burden of choice on the language user to decide which symbol to use and an additional load on the reader to remember multiple representations of the same construct.
- *Symbol overload* exists when the same graphical symbol is used to represent different semantic constructs. Such symbols are called *homographs* (the graphical equivalent of homonyms). Symbol overload is the worst kind of anomaly as it leads to ambiguity and the potential for misinterpretation [9]. It also violates one of the basic properties of the symbol system of graphics (*monosemy [1]*), as the meaning of these symbols must emerge from the context or through the use of textual annotations.
- *Symbol excess* exists when graphical symbols are used that don't represent any semantic construct. Symbol excess unnecessarily increases *graphic complexity*, which has been found to reduce understanding of notations [23].
- *Symbol deficit* exists when semantic constructs are not represented by any graphical symbol. This is not necessarily a problem and can actually be an advantage: symbol deficit may be used as a deliberate strategy for reducing graphic complexity of notations.

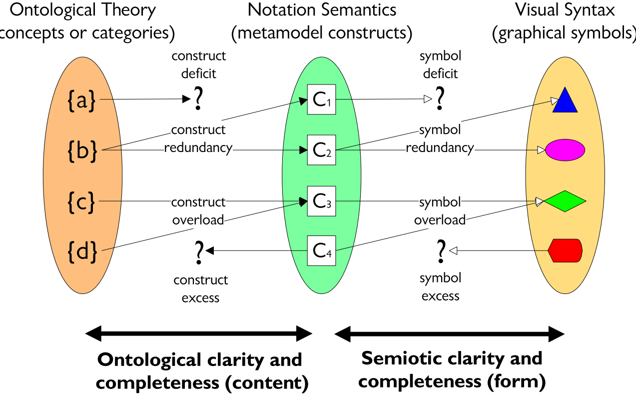

Fig. 3. Semiotic clarity represents an extension of ontological analysis to the syntactic level

If symbol deficit exists, the visual notation is said to be *semiotically incomplete*. If any of the other three anomalies exist, the notation is *semiotically unclear.*

3.2 Principle of Perceptual Discriminability

Perceptual discriminability is the ease and accuracy with which different graphical symbols can be differentiated from each other. Accurate discrimination between symbols is a necessary prerequisite for accurate interpretation of diagrams [43].

Visual Distance
Perceptual discriminability is primarily determined by the *visual distance* between symbols, which is defined by (a) the number of visual variables on which they differ and (b) the size of these differences. In general, the greater the visual distance between symbols used to represent different constructs, the faster and more accurately they will be recognised [42]. If differences are too subtle, errors in interpretation and ambiguity can result. In particular, requirements for perceptual discriminability are much higher for novices (end users) than for experts [3, 4].

Perceptual Popout
According to *feature integration theory*, visual elements that have unique values on at least one visual variable can be detected pre-attentively and in parallel across the visual field [29, 38]. Such elements "pop out" of the visual field without conscious effort. On the other hand, visual elements that are differentiated by a unique combination of values (*conjunctions*) require serial inspection to be recognised, which is much slower, error-prone and effortful [39]. The clear implication of this for visual notation design is that each graphical symbol should have a unique value on at least one visual variable.

3.3 Principle of Perceptual Immediacy

Perceptual immediacy refers to the use of graphical representations that have natural associations with the concepts or relationships they represent. While the *Principle of Perceptual Discriminability* requires that symbols used to represent different constructs should be clearly different from each other, this principle requires that symbols should (where possible) provide cues to their meaning. The most obvious form of association is perceptual resemblance, but other types of associations are possible: logical similarities, functional similarities and cultural associations.

Iconic representations
Icons are symbols which perceptually resemble the concepts they represent [27]. These make diagrams more visually appealing, speed up recognition and recall, and improve intelligibility to naïve users [3, 4]. Icons are pervasively used in user interface design [22] and cartography [44] but surprisingly rarely in software engineering.

Spatial relationships
Perceptual immediacy also applies to representation of relationships. Certain spatial configurations of visual elements predispose people towards a particular interpretation of the relationship between them even if the nature of the elements is unknown [11, 43]. For example, left-to-right arrangement of objects suggests causality or sequence while placing objects inside other objects suggests class membership (subset).

3.4 Principle of Visual Expressiveness

Visual expressiveness refers to the number of different visual variables used in a visual notation. There are 8 elementary *visual variables* which can be used to graphically encode information (Figure 4) [1]. These are categorised into *planar variables* (the two spatial dimensions) and *retinal variables* (features of the retinal image).

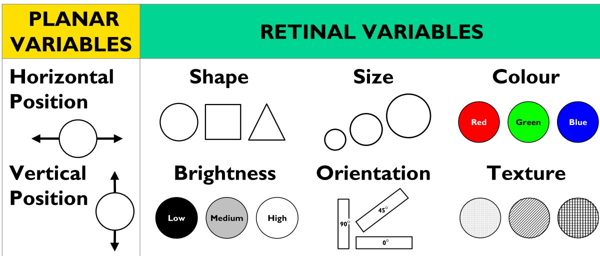

Fig. 4. The Dimensions of the Design Space: the visual variables define a set of elementary graphical techniques that can be used to construct visual notations

Using a range of variables results in a perceptually enriched representation which uses multiple, parallel channels of communication. This maximises computational offloading and supports full utilisation of the graphic design space. Different visual variables have properties which make them suitable for encoding some types of information but not others. Knowledge of these properties is necessary to make effective choices.

3.5 Principle of Graphic Parsimony

Graphic complexity is defined as the number of distinct graphical conventions used in a notation: the size of its *visual vocabulary* [23]. Empirical studies show that increasing graphic complexity significantly reduces understanding of software engineering diagrams by naïve users [23]. It is also a major barrier to learning and use of a notation. The human ability to discriminate between perceptually distinct alternatives on a single perceptual dimension (*span of absolute judgement*) is around six categories: this defines a practical limit for graphic complexity [18].

4 Evaluation of UML

In this section, we evaluate the 13 types of diagrams defined in UML 2.0 using the principles defined in Section 3.

4.1 Principle of Semiotic Clarity

The UML visual vocabulary contains many violations to semiotic clarity: in particular, it has alarmingly high levels of symbol redundancy and symbol overload. For example, of the 31 symbols commonly used on Class diagrams (shown in Figure 6

and Table 1), there are 5 synographs (16%) and 20 homographs (65%). This results in high levels of graphic ambiguity (due to homographs) and graphic complexity (due to synographs).

- *Symbol redundancy*: this is widespread in UML: examples of synographs can be found in all diagram types. For example, in Use Case Diagrams, actors can be represented by stick figures or rectangles; in Class Diagrams, interfaces can be represented by rectangles or circles. Relationships can also be represented in different ways: for example, in Package Diagrams, package relationships can be shown using spatial enclosure or connecting lines. The purpose for providing alternative visual representations is not explained but presumably to provide flexibility to notation users. However flexibility in perceptual forms is undesirable in *any* language (e.g. alternative spellings for words in natural languages) and undermines standardisation and communication.

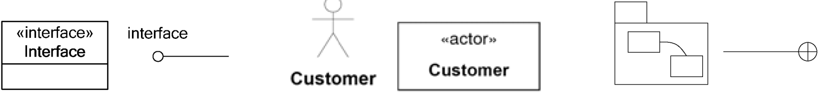

Fig. 5. Symbol redundancy: there are multiple graphical representations (*synographs*) for interfaces in Class Diagrams, actors in Use Case Diagrams and package relationships in Package Diagrams

- *Symbol overload*: this is endemic in UML, with the majority of graphical conventions used to mean different things. For example, in Class Diagrams, the same graphical symbol can be used to represent objects, classes and attributes. Different types of relationships can also be represented using the same graphical convention e.g. package merges, package imports and dependencies are all represented using dashed arrows. A major contributor to symbol overload in UML is the widespread practice of using text to discriminate between symbols (something which very few other software engineering notations do) [1].
- *Symbol excess*: the most obvious example of symbol excess and one which affects all UML diagram types is the *note* or comment. Notes contain explanatory text to clarify the meaning of a diagram and perform a similar role to comments in programs. Including notes on diagrams is a useful practice, but enclosing them in graphical symbols is unnecessary as they convey no additional semantics. Such symbols add visual noise to the diagram (an example of what graphic designers call “boxitis” [41]) and confound its interpretation by making it likely they will be interpreted as constructs.

Recommendations for Improvement

To simplify interpretation and use of the language, all occurrences of symbol redundancy, symbol overload and symbol excess should be removed:

---

[1] Symbols are considered to be homographs if they have zero visual distance (i.e. they have identical values for all visual variables) but represent different semantic constructs.

- Symbol redundancy: this can be easily resolved by choosing one of the symbols as the sole representation for a construct. Two later principles (*Perceptual Discriminability* and *Perceptual Immediacy*) provide the basis for choosing between synographs (identifying the most cognitively effective alternative).
- Symbol overload: this can be resolved by differentiating between symbols used to represent different constructs. The *Principle of Perceptual Discriminability* defines ways of differentiating between symbols.
- Symbol excess: this can be resolved simply by removing the unnecessary symbols. For example, notes can be simply shown as blocks of text so they will not be interpreted as constructs. They should also be shown using smaller font so they are clearly lower in the visual hierarchy (perceptual precedence).

4.2 Principle of Perceptual Discriminability

UML diagrams consist of two types of elements: *nodes* (two-dimensional graphical elements) and *links* (one-dimensional graphical elements). Discriminability of both types of elements are important. Figure 6 shows the node types that commonly appear on Class Diagrams, which are the most important of all UML diagrams. There are several problems with the discriminability of these symbols:

- Visual proximity: the node types (with one exception) differ on only a single visual variable (*shape*) and the values chosen are very close together: all shapes are either rectangles or rectangle variants. Given that experimental studies show that rectangles and diamonds are often confused by naïve users in ER diagrams [23], it is likely that these shapes will appear virtually identical to non-experts.
- Textual encoding: some of the symbols have zero visual distance (*homographs)* and are differentiated by labels or typographical characteristics. For example, objects are distinguished from classes by use of underlining, abstract classes by *italics* and data types by a label «data type». Text is a inefficient way to differentiate between symbols as it relies on slower, sequential cognitive processes.

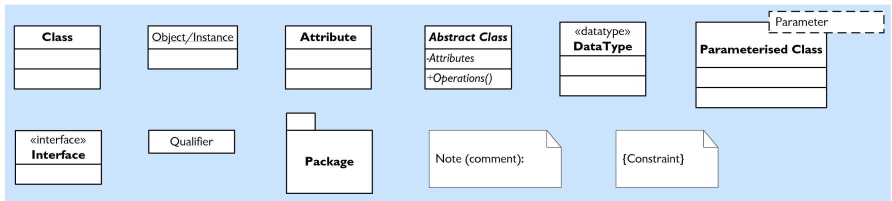

Fig. 6. Class Diagram node types: visual distances between symbols are too small to be reliably perceptible

Table 1 shows the link types most commonly used on Class diagrams. There are a number of discriminability problems with these conventions:

- Visual proximity: the link types differ on only two visual variables (*shape* and *brightness*) and the differences in values chosen are very small in many cases (e.g. closed and open arrows are visually similar).

- Non-unique values: discrimination between relationship types mostly relies on unique combinations of values (conjunctions). Only two of the links have unique values on any visual variable which precludes perceptual popout in most cases.
- Textual encoding: some of the links have zero visual distance (*homographs)* and are differentiated only by labels.

Table 1. Relationship Types in UML Class Diagrams

Aggregation	Association (navigable)	Association (non-navigable)	Association class relationship	Composition
Constraint	Dependency	Generalisation	Generalisation set	Interface
N-ary association	Note reference	Package containment	Package containment	Package import (public) «import»
Package import (private) «access»	Package merge «merge»	Realisation	Substitution «substitute»	Usage «use»

Recommendations for Improvement

There is a range of ways to improve the discriminability of UML symbols:

- Clearly distinguishable shapes: of all the visual variables, shape plays a privileged role in discrimination as it is the primary basis on which we classify objects in the real world: differences in shape generally signal differences in *kind* [2, 17, 30]. This means that shapes used to represent different constructs should be clearly distinguishable from each other. For example, packages could be shown as 3D rectangles to differentiate them from classes (Figure 7): these are clearly distinguishable from each other as they belong to different shape families. Another advantage is that their 3D appearance suggests that they can "contain" other things, which provides a clue to their meaning (*Principle of Perceptual Immediacy*).
- Perceptual popout: each symbol should have a unique value on at least one visual variable rather than relying on conjunctions of values.
- Redundant coding: discriminability can also be improved by using multiple visual variables in combination to increase visual distance [16]. Redundancy is an important technique used in communication theory to reduce errors and counteract noise [10, 33]. Figure 7 is an example of the use of redundant coding, as brightness (through the use of shading) is used in addition to shape to distinguish between classes and packages. This also facilitates perceptual popout as no other UML symbol uses shading.

- Resolving symbol redundancy: this principle also provides the basis for choosing between synographs. For example, in Use Case Diagrams, stick figures should be preferred to rectangles for representing actors because they are much more discriminable from all other symbols that can appear on such diagrams.

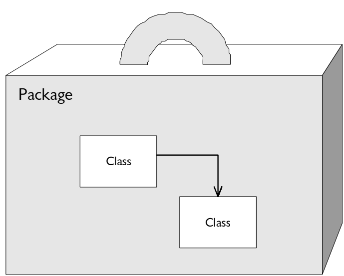

Fig. 7. Use of a 3D shape to differentiate packages from classes. A handle can also be included to reinforce the role of a package as a "container" (Principle of Perceptual Immediacy).

Redundant coding can also be used to improve discriminability of relationships in UML. In the current UML notation (Figure 8: left), the visual distance between generalisation relationships and associations is very small, even though these represent fundamentally different types of relationships (relational vs set-theoretic [12]). Discriminability of these relationships can be improved by using an additional visual variable, *vertical location (y),* to differentiate between them (Figure 8: right). Location is one of the most cognitively effective visual variables and this spatial configuration of elements is naturally associated with hierarchical relationships, which supports ease and accuracy of interpretation (*Principle of Perceptual Immediacy*) [43].

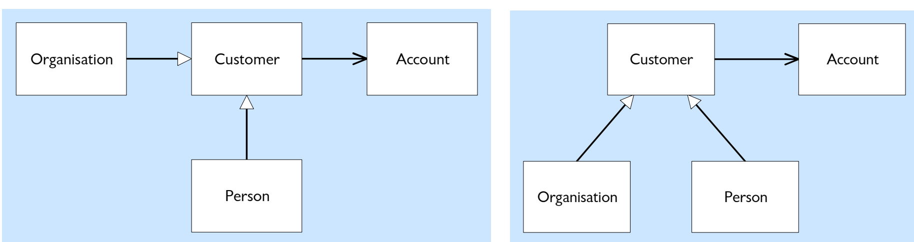

Fig. 8. Redundant coding I. Left: UML 2.0 uses only one visual variable (shape of arrowheads) to distinguish between association and generalisation relationships. Right: vertical location could be used to further differentiate generalisation relationships by always placing subclasses below superclasses.

Discriminability can be improved further by adding a third visual variable: line topology (*shape*) (Figure 9). Merged or "tree-style" line structures (where multiple lines converge into one) are also naturally associated with hierarchical relationships as they are the standard way of representing organisational charts (*Principle of Perceptual*

Immediacy). This would be even more effective if merged lines were reserved for this purpose: currently in UML, merged lines can be used to indicate generalisation sets but can also be used for many other types of relationships (e.g. aggregation, package composition, realisation) which means that it lacks any specific meaning.

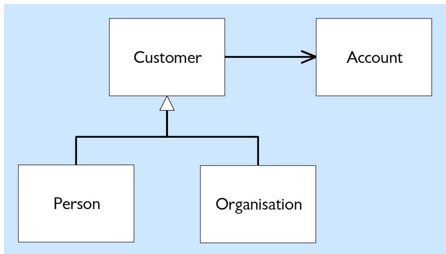

Fig. 9. Redundant coding II. Line topology (*shape*) can be used to differentiate generalisation relationships further

Discriminability can be improved further by adding a fourth visual variable: line weight (*size*) (Figure 10). Use of an ordinal variable such as size creates a natural perceptual precedence among relationship types: generalisation relationships will be perceived to be more important than other relationship types. However this is justifiable as generalisation relationships are arguably the strongest relationships of all (as they represent "family" relationships): their visual precedence is therefore congruent with their semantic precedence.

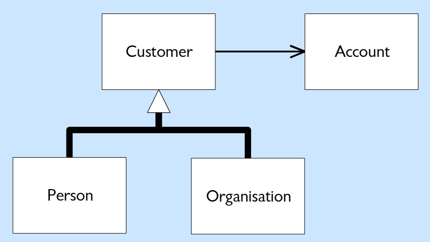

Fig. 10. Redundant coding III. Line weight (*size*) can be used to differentiate generalisation relationships further

Finally, discriminability can be improved even further by adding a fifth visual variable: colour (Figure 11).

Each step in the process progressively introduces a new visual variable, so that by Figure 11 the difference between the relationships is unmistakeable. The last three visual variables introduced represent unique values as no other UML relationship types use these values: which facilitates perceptual "popout". If merged lines were reserved for generalisation hierarchies, this too would be a unique value.

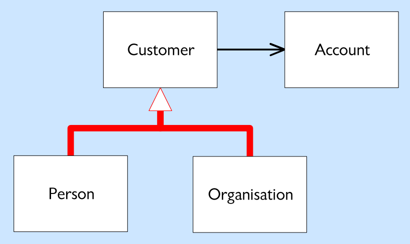

Fig. 11. Redundant Coding IV. Colour can also be used to redundantly encode generalisation relationships

4.3 Principle of Perceptual Immediacy

Like most software engineering notations, UML mostly relies on abstract geometrical shapes to represent constructs. Such symbols don't convey anything about the meaning of their referent concepts: their meaning is purely conventional and must be learnt. The only exception is Use Case Diagrams where stick figures and custom icons may be used to represent different types of actors. This may be one reason why these diagrams are more effective for communicating with business stakeholders than most other UML diagrams. UML also makes limited use of spatial relationships and relies mainly on different types of connecting lines to distinguish between different types of relationships (e.g. Table 1).

Recommendations for Improvement

Where possible, perceptually direct shapes, icons and spatial relationships should be used instead of simple "boxes and lines". For example, Figure 7 shows how a more perceptually direct shape could be used to represent packages. As another example, *whole-part relationships* are shown in a similar way to other associations and are differentiated only by a different shaped connector. This convention is not mnemonic (it is difficult for non-experts to remember which end is the "whole" and which is the "part" end) and is ambiguous because diamonds are also used to represent n-ary relationships. A more perceptually direct way to show such relationships would be to allow them to "pierce" elements (Figure 12). This makes such relationships stand out from other relationship*s (Principle of Perceptual Discriminability*) and also has a mnemonic association as the part emerges from "inside" the whole. Alternatively, a "dovetail joint" could be used.

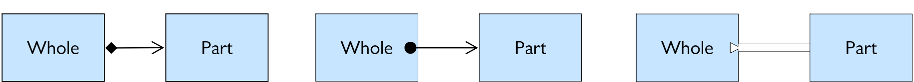

Fig. 12. "Piercing" elements (middle) conveys the concept of being "part of" in a more perceptually direct way than the existing convention (left) and makes such relationships more discriminable from other types of relationships. A "dovetail joint" (right) provides another possible option.

4.4 Principle of Visual Expressiveness

The visual expressiveness of the UML visual notations is summarised in Table 2. Most diagram types use only two of the eight available visual variables, mostly shape and brightness. Shape is one of the least efficient variables and brightness is problematic for discriminability purposes (for which it is mainly used) because it is an ordinal variable so creates perceptual precedence among symbols.

Activity Diagrams are the only diagrams that use more than two visual variables, primarily because they use both planar variables (x,y) to encode information: "swimlanes" allow the diagram to be divided into horizontal and/or vertical regions which can convey information about who performs the activity, where it is performed etc. The planar variables are the most powerful visual variables and represent the major distinguishing feature between diagrams and textual representations [1, 14]. However few software engineering notations use spatial location to convey information and it acts as a major source of noise on most software engineering diagrams. UML Activity Diagrams represent a rare example of excellence in using spatial location to encode information in the software engineering field.

Colour is one of the most cognitively effective visual variables, yet is specifically avoided in UML:

> "UML avoids the use of graphic markers, such as colour, that present challenges for certain persons (the colour blind) and for important kinds of equipment (such as printers, copiers, and fax machines)." [25]

A major contributing factor to the visual inexpressiveness of the UML visual notations is the use of text to encode information. UML currently relies far too much on textual elements to distinguish between symbols and to encode semantics (e.g. cardinalities of relationships). As a result, UML diagrams are really more textual than graphical.

Table 2. Visual Expressiveness of UML Diagrams

<table>
<tr><th>Diagram Type</th><th>X</th><th>Y</th><th>Size</th><th>Brightness</th><th>Colour</th><th>Shape</th><th>Texture</th><th>Orientation</th></tr>
<tr><td>Activity</td><td>●</td><td>●</td><td></td><td>●</td><td rowspan="13">Specifically prohibited</td><td>●</td><td></td><td></td></tr>
<tr><td>Class</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Communication</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Component</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Composite structure</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Deployment</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Interaction overview</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Object</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Package</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Sequence</td><td>●</td><td></td><td></td><td></td><td>●</td><td></td><td></td></tr>
<tr><td>State machine</td><td></td><td></td><td></td><td>●</td><td>●</td><td></td><td></td></tr>
<tr><td>Timing</td><td>●</td><td>●</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Use case</td><td>●</td><td></td><td></td><td></td><td>●</td><td></td><td></td></tr>
</table>

Recommendations for Improvement

The recommendations for improving visual expressiveness are:

- Colour: this provides an obvious opportunity for improving cognitive effectiveness of UML diagrams as colour is one of the most cognitively efficient visual variables. Differences in colour are detected faster and more accurately than any other visual variable [37, 43]. The reasons given for not using colour in UML are simply not valid: firstly, technology limitations should not drive notation design choices; secondly, graphic designers have known about such problems for centuries and have developed effective ways of making graphic designs robust to such variations (primarily through redundant coding).
- Graphical rather than textual encoding: text is less cognitively effective for encoding information and should only be used as a "tool of last resort" [24]. The more work that can be done by the visual variables, the greater the role of perceptual processes and computational offloading.

4.5 Principle of Graphic Parsimony (Less Is More)

It is not an easy task to determine the graphic complexity of the UML diagram types as the rules for including symbols on each diagram type are very loosely defined. For example, there are 6 different types of *structure diagrams* (Class, Component, Composite Structure, Deployment, Object, Package), but symbols from any of these diagrams can appear on any other (which explains their very high levels of graphic complexity in Table 3 below). This means that the graphic complexity of each type of structure diagram is the *union* of the symbols defined across all diagram types. The reason for such flexibility is hard to fathom, as the point of dividing the language into diagram types should be to partition complexity and to make the visual vocabularies of each diagram type manageable. Having such a *laissez-faire* approach places a great strain on perceptual discriminability and increases the graphic complexity of all diagrams. Table 3 summarises the graphic complexity of the UML diagram types (the

Table 3. Graphic Complexity of UML Diagrams

Diagram	Graphic Complexity
Activity	31
Class	60
Communication	8
Component	60
Composite structure	60
Deployment	60
Interaction overview	31
Object	60
Package	60
Sequence	16
State machine	20
Timing	9
Use case	9

number of different graphical conventions that can appear on each diagram type). Most of them exceed human discrimination ability by quite a large margin. A notable exception is Use Case Diagrams which may explain their effectiveness in communicating with business stakeholders.

Recommendations for Improvement

There are three general strategies for dealing with excessive graphic complexity:

1. Reducing semantic complexity: reducing the number of semantic constructs is an obvious way of reducing graphic complexity. In UML, a major contributor to graphic complexity is the level of overlap between the visual vocabularies of diagram types. To reduce this, metamodel constructs could be repartitioned so that the number of constructs in each diagram type is manageable and there is less overlap between them.
2. Reducing graphic complexity: graphic complexity can also be reduced directly (without affecting semantics) by introducing *symbol deficit*.
3. Increasing visual expressiveness: The span of absolute judgement and therefore the limits of graphic complexity can be expanded by increasing the number of perceptual dimensions on which visual stimuli differ. Using multiple visual variables to differentiate between symbols (*Principle of Visual Expressiveness*) can increase the span of absolute judgement in an (almost) additive manner [18].

5 Conclusion

This paper has evaluated the cognitive effectiveness of the 13 UML diagram types using theory and empirical evidence from a wide range of fields. The conclusion from our analysis is that radical surgery is required to the UML visual notations to make them cognitively effective. We have also suggested a number of ways of solving the problems identified, though these represent only the “tip of the iceberg” in the improvements that are possible: space limitations prevent us including more. Of all the diagram types, Use Case Diagrams and Activity Diagrams are the best from a visual representation viewpoint, which may explain why they are the most commonly used for communicating with business stakeholders [6]. Class Diagrams are among the worst – even though they are the most important diagrams – which may explain why they are commonly translated into text for communicating with users [36].

Another goal of this paper has been to draw attention to the importance of visual syntax in the design of UML. Visual syntax is an important determinant of the cognitive effectiveness of software engineering notations, perhaps even more important than their semantics. We believe that the lack of attention to visual aspects of UML is a major oversight and that as a result its visual development is lagging behind its semantic development.

5.1 Theoretical and Practical Significance

Software engineering has developed mature methods for evaluating semantics of software engineering notations (e.g. ontological analysis). However it lacks comparable methods for evaluating visual notations. The theoretical contribution of this paper is to demonstrate a scientific approach to evaluating and improving visual notations,

which provides an alternative to informal approaches that are currently used. This complements formal approaches such as ontological analysis that are used to evaluate semantics of notations.

The practical contribution of this paper is that it provides a theoretically and empirically sound basis for improving the cognitive effectiveness of the UML visual notations. This will improve communication with business stakeholders (which is currently a major weakness) as well as design and problem solving performance. Cognitive effectiveness supports both these purposes as it optimises representations for processing by the human mind.

5.2 UML 3.0: A Golden Opportunity?

The next release of UML represents a golden opportunity to redesign the visual notations in a cognitively optimal manner. The existing UML notations have been developed in a bottom up manner, by reusing and synthesising existing notations. However ideally, visual representations should be designed in a *top-down manner* based on a thorough analysis of the information content to be conveyed: form should follow content [31]. The UML metamodel provides a sound and now relatively stable foundation for doing this. Of course, such change will need to be handled carefully as practitioners familiar with the existing notation will resist radical change. However they may be more open to changes if they have a clear design rationale grounded on theory and empirical evidence, something which is currently lacking from the UML visual syntax.

References

1. Bertin, J.: Semiology of Graphics: Diagrams, Networks, Maps. University of Wisconsin Press, Madison (1983)
2. Biederman, I.: Recognition-by-Components: A Theory of Human Image Understanding. Psychological Review 94(2), 115–147 (1987)
3. Britton, C., Jones, S.: The Untrained Eye: How Languages for Software Specification Support Understanding by Untrained Users. Human Computer Interaction 14, 191–244 (1999)
4. Britton, C., Jones, S., Kutar, M., Loomes, M., Robinson, B.: Evaluating the intelligibility of diagrammatic languages used in the specification of software. In: Anderson, M., Cheng, P., Haarslev, V. (eds.) Diagrams 2000. LNCS, vol. 1889, pp. 376–391. Springer, Heidelberg (2000)
5. Cheng, P.C.-H., Lowe, R.K., Scaife, M.: Cognitive Science Approaches To Understanding Diagrammatic Representations. Artificial Intelligence Review 15(1/2), 79–94 (2001)
6. Dobing, B., Parsons, J.: How UML is Used. Communications of the ACM 49(5), 109–114 (2006)
7. Gehlert, A., Esswein, W.: Towards a Formal Research Framework for Ontological Analyses. Advanced Engineering Informatics 21, 119–131 (2007)
8. Glinz, M., Berner, S., Joos, S.: Object-oriented modeling with ADORA. Information Systems 27, 425–444 (2002)
9. Goodman, N.: Languages of Art: An Approach to a Theory of Symbols. Bobbs-Merrill Co., Indianapolis (1968)

10. Green, D.M., Swets, J.A.: Signal Detection Theory and Psychophysics. Wiley, New York (1966)
11. Gurr, C.A.: Effective Diagrammatic Communication: Syntactic, Semantic and Pragmatic Issues. Journal of Visual Languages and Computing 10, 317–342 (1999)
12. Harel, D.: On Visual Formalisms. Communications of the ACM 31(5), 514–530 (1988)
13. Jonkers, H., van Buuren, R., Hoppenbrouwers, S., Lankhorst, M., Veldhuijzen van Zanten, G.: ArchiMate Language Reference Manual (Version 4.1). Archimate Consortium, 73 (2007)
14. Larkin, J.H., Simon, H.A.: Why a Diagram is (Sometimes) Worth Ten Thousand Words. Cognitive Science 11(1) (1987)
15. Lee, J.: Design Rationale Systems: Understanding the Issues. IEEE Expert 12(3), 78–85 (1997)
16. Lohse, G.L.: The Role of Working Memory in Graphical Information Processing. Behaviour and Information Technology 16(6), 297–308 (1997)
17. Marr, D.C.: Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. W.H. Freeman and Company, New York (1982)
18. Miller, G.A.: The Magical Number Seven, Plus Or Minus Two: Some Limits On Our Capacity For Processing Information. The Psychological Review 63, 81–97 (1956)
19. Moody, D.L.: Evidence-based Notation Design: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Transactions on Software Engineering (under review, 2009)
20. Moody, D.L.: Review of ArchiMate: The Road to International Standardisation, Report commissioned by the ArchiMate Foundation and BiZZDesign B.V., Enschede, The Netherlands, 77 pages (2007)
21. Moody, D.L., Mueller, R.M., Amrit, C.: Review of ORES Methodology, Notation and Toolset, Report commissioned by Egon-Sparenberg B.V., Amsterdam, The Netherlands, 53 pages (2007)
22. Niemela, M., Saarinen, J.: Visual Search for Grouped versus Ungrouped Icons in a Computer Interface. Human Factors 42(4), 630–635 (2000)
23. Nordbotten, J.C., Crosby, M.E.: The Effect of Graphic Style on Data Model Interpretation. Information Systems Journal 9(2), 139–156 (1999)
24. Oberlander, J.: Grice for Graphics: Pragmatic Implicature in Network Diagrams. Information Design Journal 8(2), 163–179 (1996)
25. OMG Unified Modeling Language Version 2.0: Superstructure. Object Management Group, OMG (2005)
26. Opdahl, A.L., Henderson-Sellers, B.: Ontological Evaluation of the UML Using the Bunge-Wand-Weber Model. Software and Systems Modelling 1(1), 43–67 (2002)
27. Peirce, C.S.: Charles S. Peirce: The Essential Writings (Great Books in Philosophy). Prometheus Books, Amherst (1998)
28. Petre, M.: Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. Communications of the ACM 38(6), 33–44 (1995)
29. Quinlan, P.T.: Visual Feature Integration Theory: Past, Present and Future. Psychological Bulletin 129(5), 643–673 (2003)
30. Rossion, B., Pourtois, G.: Revisiting Snodgrass and Vanderwart's object pictorial set: The role of surface detail in basic-level object recognition. Perception 33, 217–236 (2004)
31. Rubin, E.: Synsoplerede Figuren. Gyldendalske, Copenhagen (1915)
32. Shanks, G.G., Tansley, E., Weber, R.A.: Using Ontology to Validate Conceptual Models. Communications of the ACM 46(10), 85–89 (2003)

33. Shannon, C.E., Weaver, W.: The Mathematical Theory of Communication. University of Illinois Press, Urbana (1963)
34. Siau, K.: Informational and Computational Equivalence in Comparing Information Modelling Methods. Journal of Database Management 15(1), 73–86 (2004)
35. Siau, K., Cao, Q.: Unified Modeling Language: A Complexity Analysis. Journal of Database Management 12(1), 26–34 (2001)
36. Tasker, D.: Worth 1,000 Words? Ha! Business Rules Journal 3(11) (2002)
37. Treisman, A.: Perceptual Grouping and Attention in Visual Search for Features and for Objects. Journal of Experimental Psychology: Human Perception and Performance 8, 194–214 (1982)
38. Treisman, A., Gelade, G.A.: A Feature Integration Theory of Attention. Cognitive Psychology 12, 97–136 (1980)
39. Treisman, A., Gormican, S.: Feature Analysis in Early Vision: Evidence from Search Asymmetries. Psychological Review 95(1), 15–48 (1988)
40. Wheildon, C.: Type and Layout: Are You Communicating or Just Making Pretty Shapes? Worsley Press, Hastings (2005)
41. White, A.W.: The Elements of Graphic Design: Space, Unity, Page Architecture and Type. Allworth Press, New York (2002)
42. Winn, W.D.: An Account of How Readers Search for Information in Diagrams. Contemporary Educational Psychology 18, 162–185 (1993)
43. Winn, W.D.: Encoding and Retrieval of Information in Maps and Diagrams. IEEE Transactions on Professional Communication 33(3), 103–107 (1990)
44. Yeh, M., Wickens, C.D.: Attention Filtering in the Design of Electronic Map Displays: A Comparison of Colour Coding, Intensity Coding and Decluttering Techniques. Human Factors 43(4), 543–562 (2001)