Extended Abstract: Towards a Semiotics of Object- and Aspect-Oriented Design

James Noble[1], Robert Biddle[2], Ewan Tempero[3], and David Pearce[1]

[1] Computer Science
Victoria University of Wellington
New Zealand
[2] Human-Oriented Technology Lab
Carleton University
Ottawa, Canada
[3] Software Engineering
The University of Auckland
New Zealand

Abstract. Object-oriented design is based on the argument that objects in a program act as a simulation of objects in the real world. This paper will provide a semiotic account of object-oriented design patterns, treating an object as a sign comprised of some part of the real world, its realisation in the program, and the programmers intent about the program design (that the object model the world). The paper will then go on to discuss the developing discipline of aspect-orientation affects the representational discourses of object-orientation. Considering the semiotics of object-oriented design can address these quetsions, both helping programmers design their programs, understand the way their designs "work", and informing the general philosophy of computer science.

1 Introduction

An object-oriented design is a "*description of communicating objects*" [13, p.3]. In this paper, we will provide a semiotic account of object-oriented design. Semiotics is the study of signs in society, that investigates the way meaning is carried by communication, treating communication as an exchange of signs. When semiotics began in the early years of the last century, most work was concerned with conventional signs — first speech, and then writing [11]. Since then, the scope of semiotics has widened to cover all kinds of signs, to the point where semiotics underlies much of structuralist and post-structuralist literary theory, film studies, cultural studies, advertising, and even the theory of popular music and studies of communications between animals (zoosemiotics) and within them (biosemiotics) [21].

2 Semiotics

Semiotics, as defined by Saussure [8], is the study of signs in society; where a *sign* is "*something standing for something else*" [11]. Since Saussure, semiotics has been applied to a wide range of different kind of signs, and for a range of diverse purposes [1, 7, 10–12, 6].

For the analysis in this paper, we will follow Eco [11] and adopt the Peirce's triadic model of the sign [19] shown in Fig. 1.

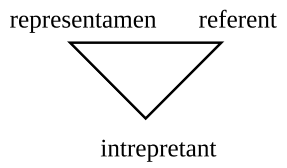

Fig. 1. Peirce's Sign

Peirce's sign is a three-part relationship between a *representamen*, a *referent* and a *interpretant*. The representamen is some phenomenon that an individual can see, hear, sense, or imagine. The referent (Peirce's term is *object*: we shall use referent in this article to avoid confusion between semiotic objects and the objects in object-oriented programs) is the a concept or entity to which the referent refers: the "something else" for which the referent stands. Finally, the interpretant is the mental concept that the representamen produces.

For example, consider the English word "chocolate" as a Peircian sign. The spoken or written word "chocolate" is the representamen; a solid compound of cocoa beans, cocoa butter, sugar and milk is the referent; the resulting mental concept of chocolate in the reader or hearer of the word is the interpretant.

3 Object-Oriented Design

A program execution is regarded as a physical model, simulating the behaviour of either a real or imaginary part of the world.

Object-Oriented Programming in the BETA Programming Language.
Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard [15]

This quote from Lehrmann Madsen et al. outlines the core principle underlying object-oriented design, that an object-oriented program simulates (or models) the world. To model a farm, for example, a program could have a Bovine class, where each object represented a cow; an Ovine class where each object represented a sheep; a Porcine class where each object represented a pig, and so on [1, 2, 17, 16]. Following a Peircian approach we can take this simulation or modelling relationship to be semiotic: that is, we can treat a program as a sign where the representamen is the set of objects and classes in the program; the referent the set of entities in the world that the program models, and the interpretant the concept that a particular program models a particular real or imaginary subpart of the world [2]. For an object-oriented program, we go further, taking a particular object in the program as a repsentamen; the part of the world that object models as a referent; and the idea that a particular object models a particular piece of the world (that the Bovine instance at memory location `0xDEADBEEF` models Daisy the cow in the field somewhere) as the interpretant of the sign. (see figure 2).

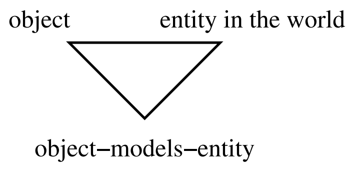

Fig. 2. An Object as a Sign

4 Implicit Signs

Unfortunately the question of what is an object in the program and an object in the world are rarely so simple. Figure 3 shows the almost generic diagram of students attending university courses. Versions of this diagram are found in many publications on object-oriented design [18, 3, 4, 20, 9]. Many students attend many courses; Courses have a course code, a title string and a teacher; students have numbers and (reluctantly, at least at our university's registry) names.

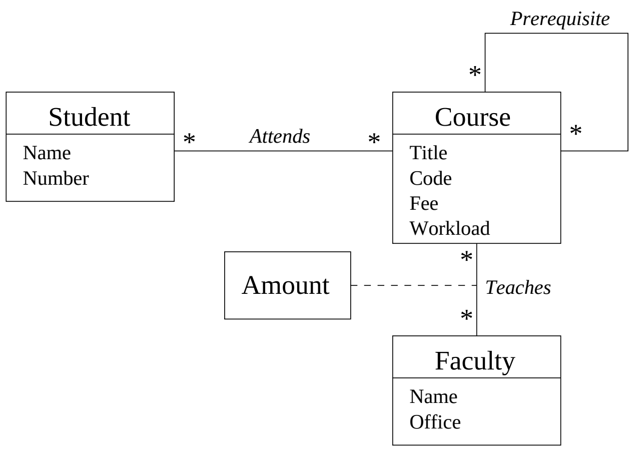

Fig. 3. A more complex design, describing part of a university

The problems arise when we consider a classical object-oriented implementation of such a design. The student class, for example, starts with easy and obvious declarations for attributes (and signatures for methods) that provide the central functionality for each class. These are relatively simple, storing the key attributes and doing basic calculations. In Java, for example, the code may be as follows:

```
class Student {
 String name;
 Integer number;
 HashSet<Course> attends;
}
```

The course class is similar — but at the bottom, we also find various pieces of code for explicitly representing and maintaining the relationships between these classes.

```
class Course {
 String code;
 String title;
 int workload;

 HashSet<Student> attendees;
 HashSet<Course> prerequisites;
 HashMap<Faculty,Amount> teacher;

 void enrol(Student s) {
   attendees.add(s);  s.attends.add(this); }
 void withdraw(Student s) {
  attendees.remove(s); s.addends.remove(this); }
}
```

This includes, for example, `HashSet` objects to record which student is attending which course (and vice versa), as well as code for enrolling and withdrawing students, whilst ensuring that all data structures remain consistent and correct. In fact, looking back at Student, things like the `attends` hash-set and the `totalWorkload` are probably more to do with relationships with courses too.

But, consider the UML design from Figure 3 again: we have three classes, each with a few attributes, and simple straightforward relationships between them. This is not well described by our simple semiotic model: first, objects like Courses, now contain HashSets and other subsidiary objects that implement intermediary data structures; second, concepts such as the "attends" relationship, that are explicit in a UML diagram such as Figure 3, are split or tangled into the implementation of other objects. Whatever the semiotic position of the `attends` HashSet, it is not representing an individual object, an individual "part of the world" that the program is modelling.

5 Aspect-Oriented Modelling

In some cases, such as that above, adopting an *aspect-oriented* approch can resolve the tension in semiotic structure of the program. In *implementation* terms, aspect-orientation

[14, 5] is generally described as including a second kind of programming construct, an *aspect*, that can include extra code and data structures into a existing class definition. From our point of view, however, an aspect, like a class or object, is simply an artifiact, a representamen, within a program, which can participate in semiotic relations with the world the program will mode.

For example, using aspects in a programming language such as AspectJ to implement the relationships from Figure 3 allows their definitions to be made quite explicit:

```
aspect Attends extends
 SimpleStaticRel<Student,Course> {}

aspect Teaches extends
 StaticRel<Course,Faculty,Amount>{}

aspect Prerequisites extends
 SimpleStaticReflexiveRel<Course>{}
```

This code introduces three aspects which correspond directly to the three relationships *Attends*, *Prerequisites* and *Teaches* in the UML design: these aspects can incorporate the code necessary to add and remove objects from the relationships. Then, the class definitions, say for the Student and Course classes, no longer need to have the extraneous code or data to implement the relationships.

```
class Student {         class Course {
 String name;            String code;
 Integer number;         String title;
}                        int workload;
                        }
```

In this way, the full paper will show that aspect-orientation can produce designs with more straightfoward semiotic properties than object-orientation: once again, a single element in the program — now either an object or an aspect instance is a representamen for a part of the real world (the semiotic referent); the programmer's interpretant being precisely this modelling relationship: that this object actually models some part of the world.

6 Conclusion

To conclude, in this paper we will demonstrate how object-oriented designs can be analysed as signs, and the worth of that analysis. Treating objects as signs provides us with an analytic framework that is based on semiotics, rather than logic, mathematics, or ad-hoc practice. Using this framework, we can address addressed a open questions about software design — particularly explicating the relationship between program designs an the external worlds they are designed to model. We hope that this framework can provide a platform for future progress in the semiotic and philosophical analyses of the practices of programming.

References

1. Peter Bøgh Andersen. *A Theory of Computer Semiotics*. Cambridge University Press, second edition, 1997.
2. Peter Bøgh Andersen. Semiotic models of algorithmic signs. In Karl-Heinz Rödiger, editor, *Algorithmik—Kunst—Semiotik. Hommage für Frieder Nake*, pages 165–211. Synchron, Heidelberg, 2003.
3. Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–282. Springer-Verlag, 2005.
4. Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
5. Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.
6. Paul Cobley, editor. *The Routledge Companion to Semiotics and Linguistics*. Routledge, New Fetter Lane, London, 2001.
7. Paul Cobley and Litza Jansz. *Semiotics for Beginners*. Icon Books, Cambridge, England, 1997.
8. Ferdinand de Saussure. *Cours de linguistique générale*. V.C. Bally and A. Sechehaye (eds.), Paris/Lausanne, 1916.
9. Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks With Uml: The Catalysis Approach*. Addison-Wesley, 1998.
10. Anthony Easthope and Kate McGowan, editors. *A Critical And Cultural Theory Reader*. Allen & Unwin, 1992.
11. Umberto Eco. *A Theory of Semiotics*. Indiana University Press, 1976.
12. Andrew Edgar and Peter Sedgwick, editors. *Key Concepts in Curtural Theory*. Routledge, New Fetter Lane, London, 1999.
13. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
14. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *ECOOP Proceedings*, 1997.
15. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
16. James Noble and Robert Biddle. Patterns as signs. In *ECOOP Proceedings*, 2002.
17. James Noble, Robert Biddle, and Ewan Tempero. Metaphor and metonymy in object-oriented design patterns. In *Proceedings of Australian Computer Science Conference (ACSC)*. Australian Computer Society, 2002.
18. David Pearce and James Noble. Relationship aspects. In *To appear in AOSD'06*, 2006.
19. Charles Sanders Peirce. *Collected Papers*. four volumes. Harvard University Press, 1934–1948.
20. Rob Pooley and Perdita Stevens. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.
21. Thomas A. Sebeok. Nonverbal communication. In Cobley [6], chapter 1.