Cohesion and Coherence in Programs

PETER BØGH ANDERSEN, DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF AALBORG.

1. Introduction

I first met Mihai Nadin in 1996 where we arranged a seminar on informatics and semiotics at Dagstuhl (Germany) together with Frieder Nake (Bremen), although I was acquainted with his writings, in particular Nadin (1988) which I had used for teaching purposes for several years.

Among his many interests were the triad computers, aesthetics, and semiotics, which also happened to belong to my favorite obsessions. The motivation for this particular assemblage was a particular view on computers, namely that they were a new medium and, consequently, that the aesthetic aspect was as important in computers as in any other medium. However, in the early eighties, computers belonged to the realm of the technical and natural sciences, whereas aesthetics was copyrighted by the humanities. Furthermore, the technology of that time did not appear as a medium, but rather as a tool or an automaton. However, the later advent of multimedia and virtual reality and the explosion of the World Wide Web changed this situation, and today everybody talks about the “computer medium” as something self-evident.

We had both hit upon semiotics and the concept of sign as a possibility for transgressing the borderlines between the faculties:

There is a physical reality to any sign (...), and there is a mental process associated with it. Sometimes the balance is inclined towards the physicality of signs, a other times towards the mental aspects.  Nadin 1993: 222.

This duality makes semiotics an inherently interdisciplinary field which enables principled reasoning across the faculty borders. And this interdisciplinarity is necessary for educational as well as for developmental purposes:

In addition, as already stated, the aesthetics cannot be improved after the VR context is created, i.e. after decisions regarding the nature of representations used have been made. They have to grow together, intertwined, in order to facilitate the much desired effectiveness of the experience.  Nadin 1995: 179.
Thus, there is ample motivation for applying a semiotic perspective to the computer medium. In 1988 Nadin suggested a semiotic analysis of the user interface of computers as displayed in Fig. 1.

![Fig. 1. The interface sign.](image)

His interface concept was very broad, and in particular included programming environments too; programmers were also users of computer systems.

Fig. 1 is fairly intuitive and easy to use, but this was definitely not the last word about computer based signs. They had many surprises in store. It turned out that computers are more than just a new tool or a new medium. As the Danish inventor of Algol, Peter Naur early realized, a computer program can be said to embody a theory, and programming shares many features with theory-building. This idea of an embodied theory lies at the heart of Nadin’s concept of “computational design”:


But how should we understand the notion of “embodied theories”? In my opinion, the concept only makes sense if we go beyond the end-user interface and delve into the interior of the program, i.e. begin to analyze the program text as a sign complex. If a system is to count as a theory, its program text must be interpretable as the kind of statements a theory is made up of. It must contain representations of general laws that can be applied to representations of “facts” (whatever they may be). In addition, as is normal in theories, these laws must be supported by empirical evidence and they must be consistent. There are some programming languages that are very easy to interpret in this way, e.g. PROLOG. In PROLOG one can write a theory consisting of impli-
computational signs, represent facts by means of assertions, and query whether some new fact follows from the old facts according to the theory:

Theory:
\[ \text{mortal}(X) :- \text{human}(X). \]

Facts:
\[ \text{human}(\text{socrates}). \]

Queries:
\[ ?- \text{human}(\text{socrates}). \]
\[ \text{yes} \]
\[ ?- \text{mortal}(\text{socrates}). \]
\[ \text{yes} \]
\[ ?- \text{mortal}(\text{zeus}). \]
\[ \text{no} \]
\[ ?- \]

But in order to use a theory properly, one must be able to assess its validity and possibly change it if it is judged false. Consequently, if a theory is embodied in a computer, the user must be able to “read” the system as a theory about some topic: the inner workings of the system must consist of sign-processes that are interpretable to the user. This is not a requirement that is normally demanded from systems (although PROLOG provides an explanation of its reasoning, which is also standard in expert systems, cf. Jackson 1990: 314 ff.). Since this is possibly a point where Nadin and I disagree slightly, it is completely in Nadin’s spirit to spend the rest of this paper on this issue: can we use semiotics to understand program texts and program executions as sign processes?\(^1\)

2. Interface and program

The first task is to relate the interface signs from Fig. 1 to the program text, and this is in fact very easy (Andersen, Hasle & Brandt 1997). Consider Fig. 2 that contains Fig. 1 as its lower part.

The program text is a representamen that denotes a — sometimes infinite set of — state changes in the computer. The interpretant relating the program text to the set of state-changes is the semantics of the programming language. This interpretant comes in two varieties: an intentional one where the designer of the programming language describes his intentions with the various constructs, and a causal one which is a computer program (a compiler or interpreter) that mechanically forces the machine to undergo the state-changes specified in the intentional interpretant.

---

\(^1\) The rest of the paper is inspired by a joint work with Frieder Nake, Bremen, Germany.
Although this is indeed a part of the semiosis, inspection of real programs reveals a much richer picture. Good program texts turn out to be made up of levels, only a few of which are actually interpreted as referring to the machine’s state changes! In fact, programmers prefer to get away from this machine-reference as quickly as possible, and to this end they have invented a repertoire of interesting semiotic processes. The main tendency is to change the indirect (via the state-changes of the machine) reference to the domain into a direct reference.

An interesting question is to which degree $I_1$ influences $I_2$, i.e. how much the interpretation of the programming language influences the possible interpretations of the interface. On the one hand, it is possible to rewrite the program completely, e.g. give it better structure, without changing the interface at all. On the other hand, there is no doubt that in some cases concepts from the program text migrate into the interface. This is clearly the case with object oriented programming (a way to structure the program) and object oriented interfaces (a way to structure the interface).

The question is interesting when we treat computers as media. The possible influences from programming languages to interface is identical to the constraints of the computer medium. Any medium has constraints that make some meanings easier to formulate than others. For example, film is a narrative medium, and it is very difficult to make film whose main purpose is phi-
losophical arguments or pure descriptions. In order to be worth viewing, the movie must contain some element of action.

3. Interpretability

The first question to ask is: are program texts really texts? In order for something to be a text, it must have cohesion and coherence. Cohesion means that the text refers to the same recurrent set of objects by means of nouns and pronouns (Halliday 1977, Togeby 1993.vol. I: 268).

Cohesion occurs where the interpretation of some element in the discourse is dependent on that of another. The one presupposes the other, in the sense that it cannot be effectively decoded except by recourse to it. When this happens, a relation of cohesion is set up, and the two elements, the presupposing and the presupposed, are thereby at least potentially integrated into a text. Halliday 1977: 4

Typically, a new object is introduced in one sentence, and commented upon in the next. Texts without cohesion seem not to be about the same world from sentence to sentence.

Coherence is a different concept. Coherence occurs when the meaning of the text can be arranged in simple symmetrical structures within a limited set of dimensions. The set of dimensions is normally called the isotopy of the text, because they are meanings that recur as a part of many sentences and words.

Most words are ambiguous, possessing a kernel meaning and many possible isotopies. For example, the word “high” has the kernel meaning “at the positive end of some scale”. The dimension in which the scale lives, is, however, variable and represents the possible isotopies of the word: spatial, social, mental, etc. When the word co-occurs with other words, the possible isotopies of the individual words mutually select the ones they share, and this becomes the actual isotopy of the text. In “high spirits” it is “mental”, in “high mountain” it is spatial, and in “high society” it is social.

Isotopies are structured according to symmetries and oppositions (on the tendency of language to form symmetries and oppositions, see Aitchison 1995). For example, in the spatial isotopy, “high” is opposed to “low” on the vertical dimension, and both are opposed to “back” and “front” that lie in the depth-dimension.

The preference for symmetries and oppositions is also known in well-structured programs. For example, if we have saving a file, we also have the opposite, opening one. If we can create a file, we can also delete it, if we can lock it, we can also unlock it, and if we can move the file into a directory we
can also move it out. A program text describing these facilities is cohesive since it is concerned with the same objects, files. In addition, it is coherent since it deals with a small set of isotopies, namely existence, access and location that are heavily interdependent: only if it exists, it can be located, and only if it can be located, it can be opened, and only if it is opened and is not locked it can be accessed. Finally, the structure of the operations exhibit symmetries and oppositions.

In summary, for a text to be cohesive it must deal with a small set of recurrent objects; for it to be coherent, it must select a few isotopies and structure them according to oppositions and symmetries.

Let us illustrate these ideas with the simple toy student registration system shown in Fig. 1. The system allows the secretary to define a new course, enter and delete students, and enter and correct marks. It can print out sheets with the students, their marks, and the average of the marks.

In order to be useful, the program has to specify features of four very different worlds:

1 The administrative world of use: The purpose in this world is for the user to apply mice and keyboards to perform student administrative actions in a correct way. The objects are: keyboard, mouse, disk, printer, and screen objects. The actions are: typing, clicking and selecting/deselecting. Important isotopies are: modality (an operation is possible or impossible) and efficiency (an operation succeeds or fails).
2. **The social world of the domain:** The purpose of this world is to regulate the social status of the students. The objects are: *students, courses, and marks*. The actions are: *enroll, withdraw, pass, flunk, grade*. If marks are required to have a particular average, the *average mark* is important.

3. **The typographical world of data:** The purpose of this typographical world is to edit data that refer to the domain world. In the scripting language employed, its objects are: *texts, lines, items, words, and characters*. If calculations are performed, the distinction between *numbers* and *non-numbers* is relevant. Important properties are: *greater/smaller* than, *identical* to. Actions include: *adding, deleting, inserting, comparing*.

4. **The world of the operating system.** The purpose of this world is to locate data and transfer it between two locations: the *persistent* location on the disk, and the *transient* location in RAM. The objects are: *files, directories, filenames, paths*. The actions are: *creating, deleting, reading and writing files*.

This means that in order to write a cohesive text, we have to divide the program text into four sections, each with their own objects and interpretants. However, as appears from Fig. 2, the language only allow us to directly describe (3), the typographical world of the data, and (4), the world of the operating system. What about world (1), the administrative actions of the secretary, and (2), the social world of the students?

According to Fig. 2, what we basically do in programming is to describe changes of data that signify objects and events in the domain of the program.

A typical data structure in the program is

- Doe, John, None
- Marx, Groucho, 13
- Monroe, Marilyn, 11
- Steward, James, 9

A typical data change is shown in Fig. 4. In the social domain this change means that “John Doe gets the mark 7”.

![Fig. 4. John Doe gets the mark 7.](image)

In the scripting language this process can be described as shown in Code 1.

```plaintext
On RecordMark aStudent, aMark
global Studentrecord
```
Code 1. Defining the action of recording a mark.

Code 1 describes the event as a typographical event consisting of inserting a number into an item of a line. If this is not typography, I do not know what typography is! We could of course write the whole program in this way, as typographical changes of the data, but such programs are very difficult to read and debug. The reason is that errors are often defined in relation to the domain (here a social one) whereas the text is about typography. Therefore various types of semioses have been invented in order to coerce the program to refer to other domains.

One such technique can be called simulation. It consists in defining a mapping $F$ that maps operations and states of one world — the source world — onto operations in another world — the target world. If we can do this, then we can stay in the target-world inside a section of the program, and do not need to be concerned with the source-world that is treated elsewhere in the text. The target world can be made to refer to a set of related objects that are different from the source world, and in this way can be made cohesive.

The mapping itself can be accomplished in various ways, e.g. by declaring functions and procedures whose names can be interpreted by means of the target-world interpretant, but whose implementation refers to the source-world. The run-time mechanism that replaces a function call by its declaration, transferring variable values from the former to the latter, is the reverse of the $F$-mapping, and it effectively reduces the target world to events and objects of the source world.

In order to do this in our case, we define a mapping between the typographical world and the social world of the students and we define the social objects, relations, and actions in terms of the typographical world.

The concept of a “mark” according to Danish legislation:

```haskell
Function IsMark aMark
    if aMark = 0 then return true
    if aMark = 3 then return true
    if aMark = 5 then return true
    if aMark = 6 then return true
    ...
    return false
End IsMark
```

Code 2. Typographical definition of “a mark” according to the Danish system of marks.

The relation “the marks of a student”:
Function TheMarksOf aStudent
   global Studentrecord
   Return item 3 of line aStudent of Studentrecord
End TheMarksOf

Code 3. Typographical definition of the social concept “the marks of a student”

The social action of grading:

function Grade aStudent, aMark
   if IsMark(aMark) then
      RecordMark aStudent, aMark
      Return “success”
   else
      return “failure”
   end if
end Grade


Having mapped the world of examinations and courses into the typographical world, we can stay in the former and use the appropriate names such as students, courses, marks, enroll, withdraw, pass, flunk, and grade, as shown in code 5:

If Grade(theStudent, aMark) = “success” then
   if theMarksOf(theStudent) < 6 then
      flunk theStudent
   else
      pass theStudent
   end if
end if

Code 5. Grading in the social world.

where Flunk and Pass will also map to the typographical world.

What happened? As shown in Fig. 5, the text contains a long series of signs that are related in a special way: the object of a “higher” sign works as the representamen of the “lower” sign. For example the definition of “Grade” contains the procedure “RecordMark”, and in this sense “Grade” can be said to refer to “RecordMark”. In the actual execution, the word “Grade” is in fact replaced by its definition, including “RecordMark”. The relationship between RecordMark and “put 7 into item 3 of line 1 of StudentRecord” is the same. The latter denotes the data change shown in Fig. 4, and the data-change denote the social event of passing exam. Such chains of representamens and objects give rise to a “short-cut” sign whose representamen is the first representamen, and whose object is the last object in the chain.
This mechanism is basic in creating cohesive program texts, but others exist, many of which are known as design patterns in computer science. Design patterns are simple recurrent and meaningful configurations of objects and protocols for their interaction.

In simulation, the source and target worlds are disjunct, sharing neither objects nor actions. However, some worlds can be more cohesive.

Consider for example the world where secretaries are using the system, compared to the social world of students. In both cases we deal with humans and their actions, only the actions differ. The actions in the social world are performative actions that create social obligations and rights when performed with a person with the right authority. In our case, when a student is enrolled he has a right to be examined in the course, and his teacher has the authority to give him marks bequeathing him new rights.

The secretary, however, does not have the right to give marks to the student, but can physically enter marks into his record if in possession of a signed exam form. Thus, the physical actions of the secretary are representa-mens signifying the social actions of enrolling and marking. The interpretant warranting the validity of this sign includes the signed exam forms.
Thus, on top of the social world of the students, we can define a new world of using the system, where physical use of controls and displays are coupled to social actions as representamen to object. This coupling inside the program text is clearly derived from the sign-relations outside the program between the actions represented by the program text. Code 6 describes two processes in the administrative use-world. The first procedure specifies how a mouseup from the secretary is to be interpreted, namely as a command to enter a mark. The description of EnterMark checks whether the secretary has selected a student on her screen: if not she is advised to select one, otherwise her action is interpreted as signifying the social action of grading.

```plaintext
on mouseUp
  EnterMark
end mouseUp

on EnterMark
  global selectedStudent
  switch true
    case selectedStudent is empty
      answer "You must select a student first" with OK
      exit switch
    case selectedStudent is not empty
      if Grade(selectedStudent, fld Mark) = "failure"
        answer "The number is not a mark" with OK
      end if
      exit switch
  end switch
end EnterMark
```


Note the we have defined two kinds of action failures: one type was defined in Code 4 in the social world of students and concerned using an illegal number as a mark. This failure is a failure to obey the social conventions of the Danish exam system, and therefore belong to the social world.

The other type of action failure pertains to erroneous operation of the interface, namely forgetting to select a student item before pressing the “Enter Mark” button. This action and its failure does not belong to the social world of students, but to the administrative use situation, and therefore it is defined here.

Thus we capture errors of marks (IsMark) in the social world where this makes sense, and errors of handling the interface (selectedStudent is not empty) in the use-world part.

4. Conclusion
In this paper I have exemplified two types of semioses programmers use, and the reason for using them, namely to make program texts more cohesive in terms of the different worlds referred to by the program. It is my conviction that as systems become more complex and increase their contents of “knowledge” and “theory”, designers must make access to this embodied knowledge easier accessible to users. As it is now, users are like readers presented with a thick book on a theory and its applications but only allowed to read the table of contents, the rest of the pages being glued together and besides written in Chinese.

This state of affairs is not equally problematic in all domains of application. The private user of a word processing program may not care, if he can afford to send for a specialist each time his system breaks down. However, the doctor using an expert system must know the arguments for the system’s diagnosis since the treatment is ultimately his responsibility, and the captain in the middle of the Atlantic Ocean cannot call for a repairman but still has to make decisions in the case of a system breakdown.

References


