Designing for palpability: Workshop at Pervasive 2007, 13-16 May 2007, Toronto, Canada. http://www.ist-palcom.org/palpable-pervasive-2007

Inspecting Abstractions as a Means to Palpability of Programs

Mads Ingstrup and Jesper Wolff Olsen

Department of Computer Science, University of Aarhus, Denmark

Abstract. Palpability comprises both mental and physical apprehensibility. We describe how reflection can be used enable programs to be self-documenting. We describe how programs are made of abstractions, how such abstractions can be inspected and how they can be meaningful and not-too-complex for users. We exemplify our approach using the architectural concept of a connector.

Keywords: breakdown, computational reflection, ubiquitous computing, palpable computing, architectural connection.

1 Introduction

The word palpable traditionally comprises both being perceptible by one or more of the five senses, as well as the notion mental apprehensibility, that is, being easily perceived by the mind (OED, 2007). Here we take outset in a quite literal acceptance of this meaning and argue how reflection can help make programs themselves perceptible to our senses and mind alike. We exemplify the approach by considering reification of connections, both networked and within the same device.

The paper is organized as follows. In the next section we take a look at the notions of *abstraction* and *concept*, and how they relates to programs. We then use these notions to characterize the technique of computational reflection in light of our agenda for using it to make programs perceptible and understandable. We argue how reflection enables the possibility for self-documentation in terms of the abstractions built into a program. Next we address the problem of how a program can allow such self-documentation to be made sense of, and how the abstractions built into a program can help manage complexity. We then turn to describing the prototype connector implementation that is under development, before concluding.

2 Abstractions

An abstraction is like a concept. A concept has intension and extension. The intension of a concept is what it means, e.g. the intension of tiger is what it means to be a tiger. The extension of a concept such as tiger is about which things are tigers [Mandler, 1997]. In the object oriented programming tradition these notions are made explicit by

having classes correspond to, or model, concepts. The extension is then the set of objects made from that class and the intension is the methods and fields it has [Madsen et al, 1993]. We will argue that intension ought to be construed slightly broader, defined not only by the programmed functionality, but also by the context of use.

Computer programs are described with abstractions. Examples of abstractions are class, object, component, connector, while-loop, variable and interface. Abstractions are connected to concepts. A particular subset of the data in memory and in the processor may be regarded as an object. However a subset of this data can only be regarded as an object if it has certain properties, just like a particular subset of atoms in the world can only usefully be regarded as a cup or monkey if this subset obeys certain properties.

A system can be understood at different levels of abstraction. But the abstractions are built into the system. A car is in the same way best understood in terms of a frame, rims, engine, tires etc. These abstractions are at a very high level compared to that of atoms, but they are not randomly chosen because they are, so to speak, built into it. If a car breaks down, the error can be described at different levels of abstraction as well. At one level, it is simply "the car broke down", at another it might be "the engine failed", and at yet another more detailed level it may be "the piston rings failed because of heavy wear". These are all perfectly adequate descriptions, and are expressed using the abstractions with which we describe and understand the subset of our environment which is the car. The same goes for a program: certain structures are built into it so it affords being understood more easily in some ways than others. To gain access to this structure at the most basic level, the programmatic one, we turn to reflection as a way to make programs document themselves and facilitate a corresponding set of levels of abstractions.

Example. Before having a closer look at reflection we will introduce a particular abstraction, the connector, which will be used as an example in the remaining sections. The concept of connectors arose in software architecture as a sister-concept to components—it is an abstraction of the interactions between components [Shaw, 1993; Shaw and Garlan, 1996]. Shaw [1993] argues convincingly, from a software engineering perspective, for using connectors as first class constructs alongside components, and several advances have been made toward that goal [Allen and Garlan, 1997; Aldrich et al, 2003]. Here, however, we are more concerned with how their explicit representation at runtime can facilitate inspection of them and consequently, the given structure of a system or program. We will not go into great detail about our model of connectors, but only sketch sufficiently rich a picture to exemplify and support the case for using introspection to make programs palpable.

As abstractions of interaction between components, connector implementations may rely on communication facilities, and therefore in general also on networking. Networking is normally designed in terms of a protocol stack that, more or less, follows ISO's 7-layer OSI model. We have modeled connectors in a way compatible with this model.

A connector consists, conceptually, of a protocol and a set of roles [Shaw, 1993]. Since the instances of connectors are often distributed, contrary to components, a connector as such exists programmatically only in the form of its roles. The roles, in our design, thus have a kind of programmatic substance. At the same time, according

to the established notion of architectural connectors, they are interfaces. We model a role as an object that implements an interface.

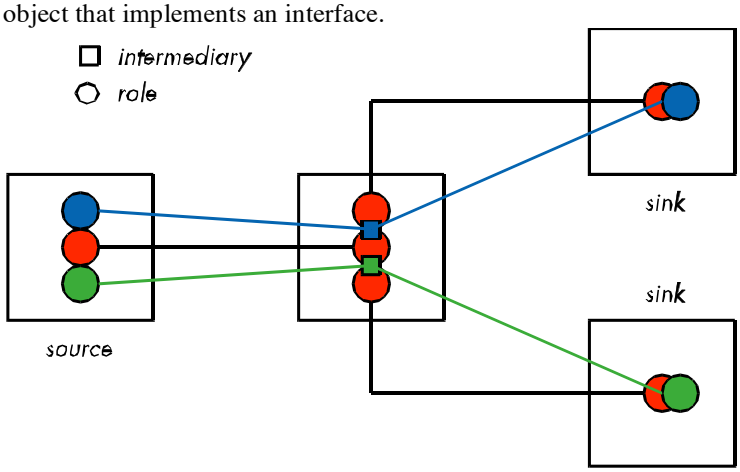

Figure 1. The diagram shows the mapping of two high-level connectors to a lower level type. There are two types of connectors: the single-hop data stream (red instances) and the multi-hop data streams (blue, green instances). The high level connectors are multi-hop binary streams. They use intermediaries to manage the routing. The roles and intermediaries are software objects while the big boxes represent nodes (hosts).

Connectors can be composed of or have *uses* relationships to other connectors. This is how we handle compatibility with the OSI model. For instance, at the data-link layer we may have simple link-connectors. (The data-link layer provides simple data-transport, but no routing, between two nodes within a particular local area network). Connectors at a higher layer, the transport layer, may then provide routing, but make use of connectors at the data link layer. In order to enable inspection of connectors, only little more information needs to be maintained than what is already present in existing networking protocol implementations. The main difference is to provide access to it through the notion of a connector.

3 Reflection and Self-Documentation

Programs manipulate data and move it around between locations in memory or I/O ports. For I/O ports this consists in reading input from or writing output to auxiliary units such as a display, keyboard or mouse. Unless a program is built so as to use these output units to document its actions, no one can know what it is doing. A very simple way for a program to document its actions would be for it to print every instruction to a screen before it is executed. However the program, or an auxiliary hardware unit, would have to be constructed to do this. To print each instruction to the screen it would translate the individual numbers interpreted as instructions into strings of text by using a table that maps numbers to instruction-names. This is, thus, a translation of the raw data into a slightly higher level of abstraction in which the abstractions used to understand the data is the set of instructions. However, it would

be useless for most users and programmers to read this directly because the level of abstraction is still too low; it would also be very inefficient.

At a higher level of abstraction the program may be written in an object oriented programming language. This means that the program is structured as classes and objects. The source-code that the programmer writes and structures using these abstractions is translated by the compiler into the machine code in the instruction set defined by the CPU or virtual machine. Although the structure and operation of the resulting program is heavily influenced by the abstractions in terms of which it was expressed (classes and objects) in the original programming language it is generally not possible to look at the resulting program and derive the source code from which it was compiled. Unlike the kind of self-documentation where each instruction is printed using a table to translate the numbers that make up program-data into instructions (a higher level of abstraction), the relationship between the program-data and its source-code is much more complex. We might say that the abstractions are not fully manifested in the source code.

For a program to be able to document itself in terms of objects and classes, the program must be programmed to do so before runtime. This is done automatically in certain object-oriented languages like Java, Smalltalk, Ruby etc., all of which support reflection. One way to achieve this reflection is for the runtime environment, in which the program is interpreted, to maintain a model of its structure and/or behaviour. In object oriented languages the model is usually only structural, because objects and classes are structural abstractions. Behavioural reflection uses different abstractions such a Lambda expressions used in LISP. We focus on reification of the connector abstraction, which belongs to the class of structural abstractions.

When a program maintains such a reflective model of itself, this model can be causally connected to the program. This means that a change in either the program or the model will cause a corresponding change in the other. If the model can be read but not changed, then it is called introspection. If it can be changed as well it is called reflection; that is, full reflection means that a program can not only introspect but also modify itself dynamically. When, in the following, we refer to reflection it is mainly in relation to the aspect of introspection. In some interpreted languages the model and ‘real program’ are not separated. Instead there is only one computational entity which is executed by an interpreter and introspectable/modifiable through a programmatic interface.

The above is a description of the technique of reflection. It is important to distinguish between introspection at the conceptual and the technical level. At the conceptual level any program, which reasons about itself in some manner, does introspection. Thus, one program may make use of a progress bar as a visualization of the progress in the process of downloading a file. This is information about the program, and as such conceptually a form of introspection. Introspection (or reflection) at the technical level is more generic. It is not for a particular purpose such as visualizing download progress. Instead it uses the general-purpose abstractions of programs, those with which programs are built by programmers, to express a reflective model. However it may of course be used to serve specific purposes. To show the progress of a download, reflection may provide access to the object that holds the buffer into which downloaded data is stored until the download is wholly complete, and to the variable on that object which represents the size of data in that

buffer. This way the progress is known if the total file-size is. When the technique of reflection is not used, the program can only visualize the progress of the download if the programmer expected that it had to do so. In the case where reflection is used, it is conceivable that a generic 'program inspection' tool could be used to show this particular aspect of the program.

The more generic nature of reflection is one of its main advantages: it decouples the mechanism of inspection from the programmer and, crucially, makes it independent of his or her ability to foresee which aspects of the program may need to be documented. It is not entirely independent, though, because the description is expressed on a predetermined level of abstraction (e.g. classes and objects; components and connectors). But in analogy with the abstractions of what makes up a cars (wheels, seats, engine etc.) we hypothesize that the particular set of abstractions that are appropriate for understanding programs are similarly stable.

Example. We have prototyped reflection of connectors in a framework manner. It works well for prototyping and experimentation, which is one of our main goals. Ideally the extensions the framework provide to basic functionality in the runtime environment (henceforth RTE) should be built into the RTE; this is desirable because most generic.

The prototype is implemented in the RTE built in the PalCom project [PalCom, 2007]. The protocol stack in this RTE is structured using *media managers* (MM), each of which corresponds to a certain choice of protocol and a certain level of abstraction cf. the OSI model. Accordingly, a TCPMediaManager provides the networking interface for the TCP protocol. We have extended a set of media managers to each implement an interface the can return all the *roles* corresponding to the level of that MM (cf. the OSI model) and its particular protocol. Thus, a TCPMediaManager is able to return a set of roles corresponding to the local endpoints of the currently open TCP connections. Since TCP is connection oriented, there is an easy mapping between *socket* and *role*, as they are almost the same. The role is different only insofar as it may reveal which other connectors it *uses*. Since connectors are only manifested in code as their roles, a *uses* relationship between two connectors means that the roles of one depend on roles of the other. Each role has a unique id of the connector it is part of and reifies. We thus reify connectors in code by their roles, and introspection of connectors is inspection of their roles and relationships among those roles.

4 Understanding Reflective Accounts

Up until this point we have argued how the basic mechanism of reflection can help achieve inspectability of programs. However, we still need to address how a user can conceivably come to understand the account a system provides of itself upon inspection.

We do not hold a reflective account as a static thing, like a photograph, a snapshot taken and subsequently interpreted by the user. Rather, the ability of a system to document itself ought to support continuous exploratory engagement with a system as

the way users pick up its meaning. This is in line with Heidegger's phenomenological account of how meaning arise through interaction, and with Gibson's theory of how perception is, fundamentally, an exploratory activity by which the subject continuously becomes aware of the environment in relation to itself and, thus, of its meanings as potentials for interaction. [Gibson 1963, 1966, 1979].

A user interface, which allows inspection of a system based on reflection, is thus like a tool that mediates the perception of a system.

As per the previous section the individuation [Smith, 1996] of the program into e.g. components, connectors and interfaces is predetermined by how it was originally programmed. Thus, the extension of the abstraction *component* within the system, that is, the determination of which things are components, is a matter-of-fact of the program's design. But the intension of the abstraction *component*, what it means to a user, is determined by that user in his or her engagement with the system. This is not to say that the programmer does not have an intended meaning of the individual parts of the program, or that the meaning cannot be largely the same for a user. Rather, it is to say that the user constructs, reconstructs, discovers or augment the meaning each part has, in the particular context of use, through exploration and interaction with that particular part.

A few elaborating comments are in order to qualify this. Firstly, note that no assumption is made that the meaning of e.g. a particular connector has to a user is not the same, or that is has a degree of continuity across different situations of use. This is very likely to be the case. Certainly, we know it is to a high enough degree that documentation is useful. The assumption that is made, though, is that our system design should allow for a variation of the *intensions* of those representations that engender the system. Secondly, field studies by Büscher [2007; in this workshop] have convinced us that this variation is particularly important to support in the case of breakdowns.

Example. How does the view that users create their own contextualized meaning through interaction apply, concretely, to our connector example? The meaning of a connector is in this case sensitive to what it connects and what that means to us. A user may be browsing the web through a connector carried by Bluetooth to a cell phone and then with UMTS onto the Internet. In this case the connector is intelligible, at the highest level of abstraction, as simply "the internet connection used by this browser". At a slightly lower level of abstraction, it may be seen as two connectors; a Bluetooth connector and the UMTS connector. Does the objective properties of the connector, that is, its observable behaviour, not define a kind of meaning unto itself, cf. the notion of intension from the OO programming paradigm as defined by the methods and fields of a class? No, not unto itself, but it certainly contributes to it. The meaning "the internet connection used by this browser" is not given fully by the program code of the particular connector instance in question, because it would be different if that instance were used for something other than connecting the particular browser to the internet.

5 Handling Complexity

While the above approach to introspection helps alleviate part of the immaterial nature of software and support practices of sense making in the first place, the issue of handling complexity still remains. The immaterial nature of the digital is an important cause for the challenge of making it physically perceptible. At the same time it provides for a plasticity that is useful in supporting multiple views and levels of abstraction, which is currently the best-known way with which to make the complexity of systems manageable.

Psychologists have long known how people shift between different modes of attention as a response to changes in their activities and environment(s) [Barg & Chartrand, 1999; Leontjev, 1983]. Indeed the breakdowns we explore here are all examples of such shifts. A breakdown like Heidegger's classic example with the hammer is characteristic of such a change in mode of attention towards the hammer, that is, from ready-to-hand to present-at-hand. Before a breakdown happens a certain state of affairs are assumed, and thus free to be left out of our immediate attention. Once this assumption breaks down, caused by e.g. the hammerhead falling off, we have an actual breakdown. Bargh & Chartrand [1999] explains how this ability to vary our mode of attention causes efficient use of the limited resources of our consciousness.

When making a system inspectable we have a problem that is similar but not identical to handling this limitation of consciousness. There is certainly more information than can be grasped at any one time. We should limit how much information about a system is available for perception at any given time, for two reasons. Firstly, it helps avoid information overload. Secondly, on a more pragmatic level, the representational medium, e.g. a computer display, may be limited in size or otherwise constraining in its capacity of how much to display. Therefore we find the problem of which parts of a system to account for and the level of detail at which to do it, is similar to the problem of which things to be conscious about, how conscious and at what level of detail. In the first case something is assumed, in the second, it is, to use a software engineering term, abstracted away.

Example. We can apply this line of thought to our concrete example of the connectors. As long as a connection between A and B (maybe both are computers, or maybe A is a computer and B is simply the internet) works fine we need not be attentive as to how data gets from A to B. As far as our consciousness is concerned we need only think of it as "a connection between A and B". As far as the software is concerned, we can handle it simply as a connector between A and B.

Next, suppose a breakdown occurs in the connector. Say the connector from A to B relies on routing data over a node, C. Now that the connection stopped working, the assumption that it is there and simply working has become invalid and we need a new way to construe things. In most systems there would be no way to know what was going on. But when using connectors we may open the connector up, inspect it, to find that where it previously used two links, from A to C and from C to B, the node C (and B for all we know) has disappeared.

6 Conclusion

A frequent cause of frustration in many systems is the failure of connections, both local and distributed, and the lack of ways to inspect them. Thus the 'mental' visibility of the problem that emerges upon a breakdown is not complemented by a corresponding physical visibility, and frustration may ensue. We considered how programs, and connectors in particular, can be made palpable by allowing inspection of the abstractions by which a program is structured.

References

1. Allen, R. and Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology,* 6(3): 213—249, 1997.
2. Anderson, S., Hartswood, M., Procter, R., Rouncefield, M., Slack, R., Soutter, J. and Voss, A. (2003). Making Autonomic Computing Systems Accountable: The Problem of Human-Computer Interaction. In *Proc 1st Intl Workshop on Autonomic Computing Systems*, 14th International Conference on Database and Expert Systems Applications.
3. Bargh, J. A. and Chartrand, T. L. The unbearable automaticity of being. *American Psychologist,* 54(7):462—479, 1999.
4. Bellotti, V., Back, M., Edwards, W. K., Grinter, R. E., Henderson, A., Lopes, C.: Making sense of sensing systems: five questions for designers and researchers. In: Terveen, L. (ed.): Proceedings of the ACM Conference on Human Factors in Computing Systems Conference. April 20-25, 2002, Minneapolis, Minnesota. p.415-422
5. Chalmers, M. (2003) Seamful Design and Ubicomp Infrastructure *Proc. Ubicomp 2003 Workshop 'At the Crossroads: The Interaction of HCI and Systems Issues in UbiComp'*.
6. Dourish, P. 1995 Accounting for System Behaviour: Representation, Reflection and Resourceful Action. *Proceedings of Computers in Context*, 1995, Aarhus Denmark.
7. Dourish, P. 2001. *Where the action is: The Foundations of Embodied Interaction*. MIT Press.
8. Gibson, J. J. *The Ecological Approach to Visual Perception*. Houghton Mifflin, 1979.
9. Gibson, J. J. *The Senses Considered as Perceptual Systems*. George Allen & Unwin Ldt., 1966.
10. Gibson, J. J. The Useful Dimensions of Sensitivity. *American Psychologist,* 18(1):1—15, 1963.
11. Leontjev, A. N. *Virksomhed, Bevidsthed, Personlighed*. Sputnik/Progress, 1983.
12. Madsen, O.L., Nygaard, K. and Møller-Pedersen, B. Object Oriented Programming in the BETA Programming Language. Addison Wesley, 1993.
13. Mandler, J. M. *Development of Categorisation: Perceptual and Conceptual Categories*. In Bremner, G., Slater, A., and Butterworth, G., editors. Infant Development: Recent Advances, pp 163—189. Psychology Press, 1997
14. OED: Oxford English Dictionary, *online edition*. www.oed.com. Accessed Jan 2007.
15. PalCom Project. www.ist-palcom.org, Accessed march 2007.
16. Shaw, M. *Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status*. In Proceedings of the workshop on Studies of Software Design, Lecture Notes in Computer Science. Springer-Verlag, 1993.
17. Shaw, M. and Garlan, D. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, 1996.
18. Smith, B. C. *On the Origin of Objects*. MIT Press, 1996.