Proceedings of CEIRE98, Special Issue of Requirements Engineering Volume 3 Number 2, pages 73-78, 1998.

A Discipline of Description
(Keynote Talk)

M. A. Jackson

Independent Consultant
101 Hamilton Terrace, London, England, jacksonma@acm.org

Abstract

Software engineers, and especially requirements engineers, are vitally concerned with describing the world. Description merits recognition as a discipline in its own right. In this talk some aspects of this putative discipline are briefly explored.

1 Introduction

Software engineering problems are problems of constructing machines to serve useful purposes in the world. The product of successful software development is a machine that interacts with its human users and with other parts of the world. The machine is physically embodied in a general-purpose computer built by hardware engineers; but the software describes the particular machine needed for the purpose in hand, and transforms the computer into that machine.

The general form of the software engineering problem is presented in [Jackson 95] and shown in Figure 1.

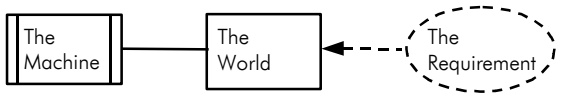

Figure 1
The General SE Problem

The striped rectangle represents the physical *machine* we must build by specialising a general-purpose computer. The plain rectangle represents the part of the *world* that interacts with the machine. The solid line connecting the two rectangles represents an interface of shared phenomena — for example, shared events and shared state. The dotted ellipse represents the intangible *requirement*, the dotted arrow indicating that the requirement is a description over the phenomena of the world. The task of the software engineer is to construct a machine such that its interactions with the world will ensure satisfaction of the requirement.

Although the end product is a description of the machine, a successful result can rarely be achieved by describing the machine alone. In general we need to make at least the following descriptions:

- The requirement $\mathcal{R}$ is an explicit description of the behaviour and properties that we want the world to have as a result of its interaction with the machine. $\mathcal{R}$ is a description in the optative mood — that is, it expresses what we would like to be true. It is a description over the phenomena of the world that are of interest to the customer of the development: it captures the purpose for which the machine is to be built and installed. In general, the requirement is not concerned primarily with phenomena at the interface between the machine and the world: the customer is usually interested in effects that are felt some distance from the machine.
- The unconditional behaviour and properties of the world that do not depend on the machine are expressed in a world description $\mathcal{W}$. $\mathcal{W}$ is an indicative description — that is, it expresses what is true of the world regardless of its interaction with the machine. It is a description over any phenomena of the world whose relationships are significant for the purpose in hand; in particular, it is not restricted to phenomena shared with the machine.
- The *specification* S describes the behaviour and properties that we want the machine to have at its interface with the world. S is an optative description. It is a description over the shared phenomena at the interface, consistent with the properties of the world and satisfiable by appropriate action of the machine.
- The *program* $\mathcal{P}$ describes the behaviour and properties that we want the machine to have, without restriction to its interface with the world. Again, $\mathcal{P}$ is an optative description. It is a description of the phenomena of the machine, including those private machine phenomena that are in the scope of the programming language but not externally observable.

These descriptions constitute a complete statement of the original problem. To demonstrate that it has been solved, we must have

$$S \wedge \mathcal{W} \rightarrow \mathcal{R}$$

That is: if the machine achieves the behaviour S at its interface with the world and the properties of the world are as described by $\mathcal{W}$, then the requirement $\mathcal{R}$ will be satisfied. Also, we must be able to construct a program $\mathcal{P}$ such that the machine can execute $\mathcal{P}$ and, by doing so, will guarantee to ensure satisfaction of the specification S.

Thus we show that execution of the program $\mathcal{P}$ will ensure the behaviour S at the machine's interface with the world, and the properties of the world described in $\mathcal{W}$ will ensure satisfaction of the requirement $\mathcal{R}$.

2 The Formal and the Informal

In any system of this kind it is important to recognise that the machine is completely formal, while the world is almost invariably mostly informal. The machine has been carefully constructed so that its fundamentally informal physical nature has been tamed and brought under control. The hardware engineers have succeeded in using continuously varying voltages to achieve entirely reliable represention of binary storage states and entirely reliable logic gate devices. This achievement allows the computer and its programs to become fully formal. That is, they can be completely and precisely described by formal finite descriptions. We can then reason formally about the computer and its behaviour, and rely on the conclusions of that reasoning.

This is why some eminent computer scientists very properly wish to treat programming in a fully formal way. The programmer's main task is to create a program — a formal description in a mathematically sound language — and to demonstrate formally that it meets its formal specification [Dijkstra 89].

However, programming is only a part of software development. Developing requirements and specifications is a larger part, and arguably more crucial. These tasks are concerned with the informal real world, in which it is impossible to give a complete and fully formal description of the purpose to be served and to demonstrate formally that a proposed specification will ensure satisfaction of that purpose. It is, however, the task of the requirements engineer to give a sufficiently formal description, and a sufficiently convincing demonstration. Description technique is concerned with finding ways to make that task feasible in the particular context of each particular development. Inevitably, this means paying serious attention to the world we are describing, and exerting ourselves to ensure that our descriptions — which are necessarily only approximations — are accurate enough for the purpose in hand.

3 Designations

Designations are a vital tool in describing an informal reality. A designation has two parts: a formal term — such as a predicate symbol and the appropriate number of dummy arguments — and a recognition rule explaining how to recognise some class of phenomena in the reality to be described. For example, in the designation:

$$\text{mother}(m,p) \approxeq m \text{ is the genetic mother of } p$$

"mother(m,p)" is the formal term and "m is the genetic mother of p" is the recognition rule.

The purpose of a designation is twofold. First, writing designations forces us to decide which phenomena our descriptions will be about. That is, we are forced to decide where we will ground our descriptions in reality. Second, it ties the formal terms we will use to denote those phenomena to a recognition rule that allows us to cash our descriptions in real world observations. A properly written designation precludes the rejoinder to any description "Well, it all depends on what you mean by *mother*." In short, it forces us to know what we are talking about.

In general, we have some choice in selecting phenomena to designate. The situation is similar to the situation that confronts anyone who gives directions to a stranger. We do not say "carry on this road for a while, then bear round towards the more attractive side, and continue until the scenery becomes fuller." We know that the unfortunate stranger will be quite unable to recognise the distance 'for a while', will not know which way to 'bear round', and will be unable to recognise either the 'more attractive side' or the 'fuller scenery'. So instead we say "continue to the second traffic lights and turn left there; then turn right at the second roundabout." Traffic lights and roundabouts are more easily recognised. If we are conscientious we will take some thought to check that the particular traffic lights and roundabouts we mean to refer to will not cause difficulty by proving hard to recognise — that the lights are not pedestrian crossing lights and the roundabouts are not mini-roundabouts or double roundabouts.

The need to choose reliably recognisable phenomena is particularly acute when the application domain has an established but unsatisfactory terminology. Notable examples are 'call' in telephony and 'flight' in airline operations. The notion of a 'call' originated in the earliest days of telephony, when only POTS — Plain Old Telephone Service — was available. A call begins when the caller lifts the phone and ends either when the callee proves to be unavailable or when the caller and callee, having been successfully connected, complete their conversation and put their phones down. But with the introduction of a 3-way calling feature the notion breaks down. Subscriber A calls B, puts B on hold, and calls C. Then A joins the two calls in a conference, and all three parties can converse together. Later A puts the phone down, leaving B and C talking. How many calls is this?

Unsatisfactory terminology can survive in systems that are partly manual, because human discretion and initiative can handle the anomalies that arise. In a more fully automated system the formal nature of the computer and its software preclude discretion and initiative: the finite software description strictly bounds the possible behaviours of the machine. This does not, of course, mean that the software developers have foreseen everything that can happen. On the contrary, developers are often surprised by the behaviour of their programs. But those programs, however ill understood by their users and creators, entirely lack the human ability to respond to new situations in new ways. The requirements engineer, then, must ensure that the system will not be surprised by the world.

4 Definitions

Designations are used to capture and name relevant phenomena of the world. For convenience in description it is usually necessary to extend the terminology by defining new terms on the basis of designated or previously defined terms. For example, having designated mother(m,p), father(f,p) and male(p), we can define sibling:

$$\text{sibling}(p,q) \triangleq \exists\, m,f \bullet \text{mother}(m,p) \wedge \text{mother}(m,q) \wedge \text{father}(f,p) \wedge \text{father}(f,q)$$

and brother:

$$\text{brother}(p,q) \triangleq \text{male}(p) \wedge \text{sibling}(p,q)$$

The essential foundation for definition is to distinguish it from assertion. If we read in a description "StockCount = cumulative total of item quantities in Issue and Receipt events", we must be able to distinguish two entirely

different meanings. In the first, the statement is an assertion that nothing irregular happens in the warehouse: there is no theft or evaporation, and no spontaneous creation of items. For this meaning there must be three designations:

Receipt(e,q,t) ≋ in event e, q items are received at time t

Issue(e,q,t) ≋ in event e, q items are issued at time t

StockCount(q,t) ≋ q items are in the warehouse bin at time t

and the assertion is that for any observation StockCount(q,t) the value of q is equal to the total of quantities q in all Receipt and Issue events occurring before t. This is an empirical statement about the world, falsifiable by observing a counterexample.

In the second meaning, the statement is a definition of the new term StockCount. There are only two designations:

Receipt(e,q,t) ≋ in event e, q items are received at time t

Issue(e,q,t) ≋ in event e, q items are issued at time t

The two designations are used to define the new term:

$$\text{StockCount}(q1,t1) \triangleq q1 = (\Sigma\, e,q,t \mid t < t1 \wedge (\text{Receipt}(e,q,t) \vee \text{Issue}(e,q,t)) \bullet q)$$

The definition says nothing about the world. It only says how the term StockCount(q,t) will be used and understood.

Definitions can be used to extract and rebuild useful meanings of unsatisfactory domain terminology. They also allow descriptions to be grounded in the world by the smallest possible number of designations. The bridge between the world and our descriptions of it becomes much more manageable because it is both precise and economical.

5 Description Structures

Usually the world and the problem requirement are too complex to be treated as single wholes. We must decompose the world into domains. In the notation of Figure 1, a decomposition of the world into three domains gives the configuration shown in Figure 2:

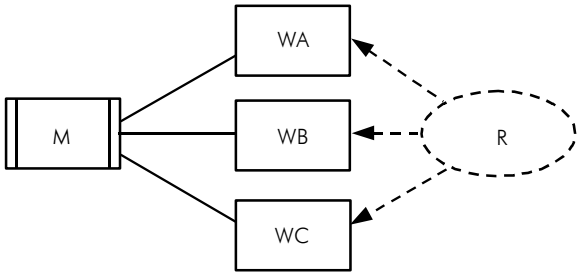

Figure 2
Decomposing the World into Three Domains

As in Figure 1, the solid lines connecting the machine M to the world denoted interfaces of shared phenomena. The diagram shows that no phenomena of interest are shared between the domains WA, WB and WC into which the world has been decomposed. In Figure 2 the world is decomposed, but the requirement R is not. The problem is still being regarded as a whole, although the world it concerns is structured into the three domains WA, WB and WC. Decomposition of the problem requirement is represented in a diagram such as Figure 3:

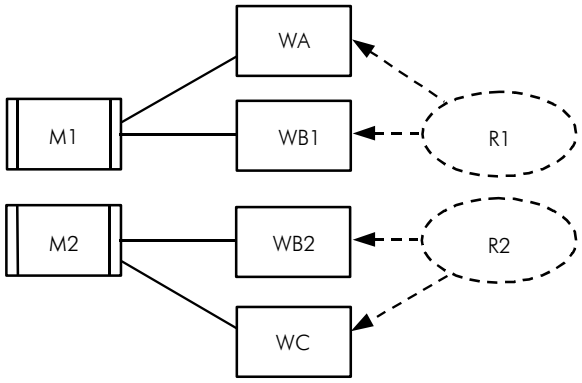

Figure 3
Decomposing the Requirement

Here the requirement R has been decomposed into two requirements R1 and R2, each corresponding to a recognised subproblem. The domain WB appears in both subproblems, but not all of its phenomena are relevant to both. The two projections WB1 and WB2 represent the two partial views of WB that are appropriate to each subproblem [Jackson 95, Jackson 96].

In some problem classes it is necessary to introduce a new domain into the problem that was not present in the original formulation. Since we are concerned with software engineering and not with physical systems engineering, the new domain will always be a part of the machine we are building, and will be constructed in software. One very common example of such a new domain is a model of a part of the given world. To provide information about phenomena that are in the past at the time when the information is produced, or are otherwise inaccessible [Balzer 82], it is often appropriate that the machine should create, maintain and use an analogic model of a given domain. Figure 4 shows such a problem structure.

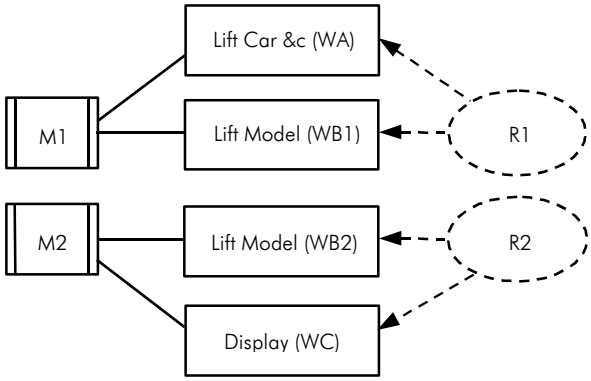

Figure 4
Introducing a Model Domain

The problem is to provide a display (WC) in a hotel lobby of the current positions of the lifts and the currently outstanding requests for service at the different floors (WA). To solve this problem it is necessary to introduce a model (WB) of the lifts. The original

problem is decomposed into two subproblems: the subproblem requirement R1 is to create and maintain the model, while the subproblem requirement R2 is to use the model to maintain the lobby display. As in Figure 3, the model has different projections in the two subproblems. The projection WB1 is concerned only with the creation and maintenance of the model; the projection WB2 is concerned only with its use to maintain the lobby display.

6 Models

It is a great misfortune in software development that the word 'model' has become so devalued. In common usage it means no more than 'description'. But in the problem decomposition of Figure 4 the model domain WB is not a description: it is an analogic model in which each relevant phenomenon of the modelled reality WA has a corresponding model phenomenon in WB. Because models are often used in solving software engineering problems, this analogical relationship, and its implications for description, are of great importance.

We may illustrate the point by a simpler example. Suppose that we need to have a model of nineteenth-century English novels. Our model, as often, will take the form of a database. Our view of the world is that each novel has a unique author; we will represent the novels and authors by N-records and A-records respectively, and authorship by a pointer from an N-record to an A-record.

The essence of the analogical relationship is that one description is true both of the reality and of the model. In predicate logic this description is:

$$\forall\, x : \mathrm{N}(x) \bullet \exists!\, y : \mathrm{A}(y) \bullet \mathrm{P}(x,y)$$

That is: for any x of which N(x) is true, there is a unique y of which A(y) is true such that P(x,y) is true. To interpret this description we must apply the appropriate sets of designations: one for the real world of authors and novels, and one for the database domain of records and pointers.

For the real world:

N(x) ≋ x is a novel

A(x) ≋ x is an author

P(x,y) ≋ x is produced by y

and for the database domain:

N(x) ≋ x is an N-record

A(x) ≋ x is an A-record

P(x,y) ≋ x points to y

It is the existence — even if it is purely implicit and unrecognised — of these two sets of designations that makes the model useful. But if they are only implicit then the model and its common description pose a danger. In writing and reading the description we may forget whether we are talking about the world, the database, or both.

Because any description is partial, there are aspects of the world that are ignored or distorted in the common description. For example, some novels have more than one author, and at least one modern novel has been produced on the internet in such a way that it has, arguably, no author at all. We are also ignoring sequences of novels, characters common to several novels, and many other features of the world of novels. At the same time we are ignoring aspects of the database world — indexes, placement of records on disk cylinders and sectors, record deletion, and record ordering. This situation poses a danger because it is very easy to develop a description without being completely clear whether we are describing the world, the database, or both. This confusion is particularly notable in object models, which often start out as a more or less conscientious attempt to describe the world, but easily drift into describing the object model that will be maintained inside the machine we are building.

7 The Procedure of Description

Even having recognised and understood the importance of making sound designations, we may still be tempted to proceed with our descriptions first, planning to return later and designate or define our terms explicitly. This temptation is especially strong when we are writing in a semi-formal notation such as those incorporated into UML. The apparently authoritative *imprimatur* of its commercially energetic sponsors tells us that use of this notation is 'best practice'. What more could we need?

Unfortunately, we need much more. Notations such as those of UML are very seductive. They make it easy to cover the screen — or a sheet of paper — with apparently meaningful symbols. But on returning later to clarify their meanings we find that it is impossible. The symbols and names that seemed so obviously meaningful when we used them prove very hard to relate to the world we are supposedly describing. Because we are unsure what the terms denote, the whole description becomes impossible to validate or criticise. Lacking explicit designations, we are are compelled to treat the descriptions themselves as if they were designations. A rough initial idea of the meaning of a term, suggested by its natural language interpretation, is fleshed out by testing it against the assertions contained in the descriptions that have been read so far. If the description fits the putative meaning, that is a partial confirmation; if not, it is the meaning that must be adjusted. As we read further, this process of testing and adjusting interpretations must be carried out, more or less simultaneously, for every term used in the description.

Obviously, reading in this way, we can not check the truth of the assertions we encounter. We are limited to the traditional feeble response: "Well, it all depends on what you mean by *mother*." An assertion that seems false leads us merely to adjust our interpretation of the terms used, rather than to challenge the assertion itself. Only when the possibilities of interpretation are exhausted can the assertion be challenged. This will not occur until either a formal contradiction is discovered or the interpretation of a term is strained to a point that is no longer credible.

Another unfortunate effect of casual use of semi-formal notations is a spurious sense that we are necessarily describing the requirement because we are using a notation purportedly designed to capture requirements. The description of use-cases is a prime culprit here. A use-case describes essentially what happens at the interface between the world and the machine, where the use-case actor interacts directly with the machine. But, as briefly discussed in Section 2, what happens at the interface between the world and the machine is the specification S, not the requirement R. To understand and capture the requirement we must study and describe

what the actors are doing while they are not interacting with the machine.

8 The Real Problem

In a wonderful book about mechanical and structural engineering [Ferguson 92], Eugene Ferguson complains that many engineering disasters have happened because modern engineers have been taught to pay too much attention to calculation and formal analysis of structures and too little to the physical reality of the world of which those structures are a part. He writes:

> "The real problem of engineering education is the implicit acceptance of the notion that high-status analytical courses are superior to those that encourage the student to develop an intuitive 'feel' for the incalculable complexity of engineering practice in the real world."

In software engineering we are not inclined to pay too much attention to 'high-status analytical courses', but we do pay a great deal of attention to techniques that are essentially notational, leaving us — like the engineers whose education Ferguson is criticising — paying too little attention to the incalculable complexity of engineering practice in the real world. Requirements are in the world, not in the machine. We must focus on them directly, and describe them conscientiously.

References

[Balzer 82] Robert M Balzer, Neil M Goldman and David S Wile; Operational Specification as the Basis for Rapid Prototyping; ACM Sigsoft SE Notes 7, 5, December 1982, pages 3-16; reprinted in New Paradigms for Software Development; W W Agresti; IEEE Tutorial Text, IEEE Computer Society Press, 1986.

[Dijkstra 89] Edsger W Dijkstra; On the Cruelty of Really Teaching Computer Science; Communications of the ACM, December 1989.

[Ferguson 92] Eugene S Ferguson; Engineering and the Mind's Eye; MIT Press, 1992.

[Jackson 95] Michael Jackson; Software Requirements & Specifications: a lexicon of practice, principles and prejudices; Addison-Wesley and ACM Press 1995.

[Jackson 96] Daniel Jackson and Michael Jackson; Problem Decomposition for Reuse; Software Engineering Journal 11,1 pages 19-30, January 1996.