Some Basic Tenets of Description

Michael Jackson

101 Hamilton Terrace, London NW8 9QY, England
jacksonma@acm.org

Abstract. Description—often referred to as modelling—is fundamental to software development. The developer should always be ready to say of each description: what subject it describes; what it says about its subject; and how it fits with other descriptions in the same development. Sometimes a very informal—even a casual—approach to these questions may be adopted. But often a more careful and explicit approach is needed. This short paper lays out some basic tenets of such an approach.

1 Introduction

The practice of software development is concerned, above all, with making and using descriptions. A program is a description of a *machine* with a desired behaviour: it describes the behaviour that a general-purpose computer will exhibit when it executes the program. A requirement is a description of desired effects in the world outside the computer: for example, that the lift will come when we press the button to summon it; that the book you have reserved will be available when you go to the library to collect it; or that an email message I send to my friend's address will find its way to his mailbox. An object diagram may describe an invariant property of the problem world: for example, that each employee has exactly one employee number. Or it may describe an invariant property of the database: for example, that *EmpNo* is the primary key of the *Employee* table.

Some software projects can be carried through in an informal way, with few explicit descriptions other than the program text and some visualisations of the program structure and execution. But often a more formal and carefully explicit approach to description is needed for all or part of a development.

2 The Problem Domain and the Machine

The first and most fundamental principle of description is to identify clearly what is being described. In software development the primary distinction is between the *Machine* and the *Problem Domain*.

The Machine is the software we build and the computer that executes it. The Problem Domain is that part of the world in which the Machine is required to bring about some effect desired by the *Customer* for whom the system is developed. For example, in a business system the Problem Domain may be the products and suppliers, the bank, the warehouse, the sales staff, the customers

of the business, and so on. In a lift control system the Problem Domain is the lift shaft, the lift car, the motor and winding gear, the buttons and lights, the floors served by the lift, and the people who use it. In a library administration system the Problem Domain is the books, periodicals and other items in the library, the library members, the staff, the library building with its shelves and desks and doors, the plastic membership cards, and all the other equipment needed to run the library[1].

Essentially, the Machine is *what must be built*, and the Problem Domain is *what is given*. Of course, this distinction is relative to the task in hand: in any decomposition of a problem into subproblems, each subproblem has its own Machine and its own Problem Domain[2].

The relationships of the Machine, the Problem Domain and the Requirement are shown in Figure 1.

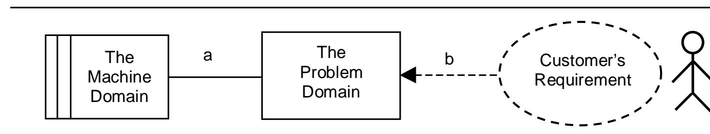

Fig. 1. The Machine Domain and the Problem Domain

The Machine is connected to the Problem Domain by an interface *a* of *shared phenomena*: in the lift system, for example, the Machine can turn the motor on by causing a *MotorOn* event, and it can sense whether a particular request button is depressed by monitoring the state of *FloorUpButton[3]*. *MotorOn* is a shared event, controlled by the Machine, and *FloorUpButton[3]* is a shared state, controlled by the Problem Domain.

The Customer's Requirement is some constraint on the Problem Domain that the Machine must enforce[3]: for example, that when the button is pressed the lift comes soon, and the doors stay open long enough for users to enter and leave the lift. The Requirement is expressed in terms of some phenomena *b* of the Problem Domain, and we may think of the Customer as observing the Problem Domain at the imaginary interface *b* to determine whether the Requirement is satisfied.

[1] We might instead call the Machine the *Solution* Domain; but we prefer the term *Machine* because it emphasises our focus on solving problems by building software machines.

[2] The decomposition of problems, and the careful identification of subproblem Machines and Problem Domains, is a central topic of [2].

[3] The fact that the Requirement constrains the domain and does not merely refer to it is indicated by the arrowhead.

There are clearly at least two distinct subjects of description here: the Machine and the Problem Domain. The Machine must certainly be described because eventually we must write the program text: that is, a description of the Machine's internal and external behaviour.

The Problem Domain must be described because we must describe the Requirement constraint, and also the domain's own, given, properties. For example, the lift domain has the property that if a *MotorPositive* event occurs followed by a *MotorOn* event then the lift starts to rise in the shaft; and if the lift car keeps rising from floor 2 it will arrive at floor 3; and when it passes a point just $6in$ below the home position of floor 3 it causes the floor sensor to close and *SensorOn[3]* becomes true.

For a description to be useful we must be completely clear about the exact meaning of each term it uses[4]. For each descriptive language, if it is well-defined, we already understand clearly the meanings of the general symbols and syntactic constructs used in every description expressed in that language[5]. But a specific description also uses specific terms, denoting phenomena of the domain it describes. It is here, in understanding the meanings of these specific terms, that clarity is particularly needed. This need, however, is often ignored, because of an unspoken assumption that all descriptions are about the Machine: we are not in doubt about the meanings of specific terms in program texts, since a program is, in effect, a formal object in which the program text itself furnishes the meanings of the specific terms it uses. By contrast, a term denoting phenomena of an informal domain outside the Machine must be clarified by an explicitly stated rule for recognising what is, and what is not, an instance of the class of phenomena denoted.

For example, in the lift system the term *ArrivesAtFloor[n]* might mean "The floor sensor at floor n flips from *off* to *on*"; or "The distance of the lift car from the home position at floor n changes from $> 6in$ to $\leq 6in$"; or "The lift car reaches the home position at floor n and the doors open." Reasoning about any description in which this term appears is likely to be pointless unless its exact meaning is explicitly stated and known.

3 Describing the Machine Is Not Enough

It is tempting to believe that a sufficient description of the Problem Domain can be given implicitly by describing the Machine. Two facts make this belief slightly plausible.

First, a full Machine description includes a description of its behaviour at its interface a with the Problem Domain. Since this is an interface of shared phenomena it seems that it can be equally well described from either side.

There is certainly some truth in this, but not enough: the Problem Domain properties at the interface are only a part of what must be described. For exam-

[4] As John von Neumann wrote[4]: "There is no point in using exact methods where there is no clarity in the concepts and issues to which they are to be applied."

[5] These are the meanings defined by a formal semantics.

ple, *SensorOn[3]* is a phenomenon of the interface a, but the exact position of the lift in the shaft is not: it is a *private* phenomenon of the Problem Domain. So in a description limited to phenomena of the Machine it is impossible to express the causal relationship between the lift position and *SensorOn[3]*. We will return to this point in Section 4.

Second, the Machine will often contain a *Model* of the Problem Domain. This Model is typically an assemblage of data type instances—objects or relations or other types—stored in the program's local storage or held in an object or relational database on disk. Its specific purpose is to reflect the state of the Problem Domain, making it always accessible to the Machine.

So it looks plausible to suppose that a description of the Model will capture, *mutatis mutandis*, the Problem Domain properties that are of interest. Again, there is sometimes some truth in this, but not often and not enough: we will return to this point in Section 6.

4 The Problem Is Not at the Interface

Developers departing reluctantly from the long tradition of their trade—in which only the Machine has been the subject of explicit description—may be comforted to believe that at least their voyage can be curtailed at the Machine's external interface to the Problem Domain. The Customer's Requirement can, perhaps, be expressed entirely in terms of *Use Cases*, each Use Case[3] embodying some short bout of interaction between the Machine and an actor, in which the actor obtains some 'observable result of value'.

In some problems this view is exactly what is needed. For example, in the design of software to control a vending machine, the Requirement is essentially expressible in terms of Use Cases. The Use Cases may be *Buy*, in which the actor is a user of the machine, *Replenish*, in which the actor is an employee of the vending company, and *Service*, in which the actor is a service engineer. If the vending machine behaves as required in each bout of interaction taken individually, then the whole Requirement is satisfied. Sequences of Use Cases need not be considered.

This approach can work well when the only Problem Domain events of significance are those that occur in the Use Cases. Nothing of interest happens except in a Use Case: essentially, the behaviour of the actors when they are not interacting with the machine does not affect the required treatment of the Use Cases. Each Use Case then approaches the Machine, so to speak, with a clean slate. Of course, an unbroken succession of purchasers may eventually empty the vending machine's stocks, and all subsequent users before the next *Replenish* instance will be disappointed. But this does not significantly complicate the treatment of the *Buy* Use Cases. Any relevant state is immediately available to the Machine in the vending machine interface.

In systems of this kind it is not necessary to look further into the Problem Domain than its interface with the Machine and the interactions that take place there in the Use Cases. But many—perhaps most—systems are not of this kind.

It is then necessary to take explicit account, and make explicit descriptions, of the properties and behaviour of the Problem Domain remote from the Machine interface.

5 Requirements Are Not Given Properties

It is essential to distinguish clearly between the properties of the Problem Domain that are given, and those which the Machine must enforce.

The given Domain Properties are those that the Problem Domain possesses regardless of the effect of the Machine. The fact that the lift can not go from floor 2 to floor 4 without passing floor 3 does not depend on the Machine's behaviour. It is built into the fabric of the Domain. The same is true of the causal chain by which motor events cause the lift to rise in the shaft. In the library it is true of the fact that the same copy of a book can not be simultaneously on loan to two different borrowers.

Requirements, by contrast, are not given but *desired* properties. They are additional constraints that must be enforced by appropriate Machine behaviour at the Problem Domain interface.

It is seriously confusing to make just one description of the Problem Domain, combining Domain Properties with Requirements. It's hard to be sure from a combined description whether a particular property is a goal of the development or an assumption that may be relied on in pursuing the goals[6].

When the Problem Domain is structured into two or more component domains, as it will be in any realistic system, not every domain will need both Domain Properties and Requirements descriptions. Essentially, Requirements descriptions are needed only for those domains that the Machine must control in some way. For example, in a system to display current prices and volumes for a stock exchange the Problem Domain may be structured as two component domains: the Stock Exchange and the Display. No Requirements description is needed—or possible—for the Stock Exchange, because the Machine only monitors that domain and does not attempt to control it. But the Display needs both a Requirements description, stipulating what must be displayed, and a Domain Properties description, stating how the Machine can bring about the required display by operations at its interface.

6 The Model Is Not the Reality

The notion of a Model of the Problem Domain (mentioned earlier in Section 3) is well known, both in theory and practice. Here the word 'Model' means a part of the Machine's local storage or database that it keeps in a more or less synchronised correspondence with a part of the Problem Domain. The Model can then act as a *surrogate* for the Problem Domain, providing information to

[6] The convention of using the word 'shall' to indicate a Requirement is an attempt to mitigate this difficulty. The cure is complete separation.

the Machine that can not be conveniently obtained from the Problem Domain itself when it is needed. For example, the library system maintains and uses a database model of the books and members, and the lift system maintains and uses a local object model of the motor state, the lift position, and the outstanding service requests.

A Model in this sense is not a *description* of anything. It is a distinct domain in itself, a designed component of the Machine Domain, that must be kept in a certain correspondence with its *Subject Domain*. Phenomena of the Model, such as data structure instances and field values, must correspond to phenomena of the Subject Domain. For example, part of the required correspondence may be that if the library book b is a copy of title t, then the *title* field of the *CopyRecord* for b must contain a pointer to the *TitleRecord* for t.

Maintaining this correspondence should be viewed as a subproblem in its own right, distinct from the subproblem in which the Model is used to produce information. The correspondence subproblem can be represented as in Figure 2.

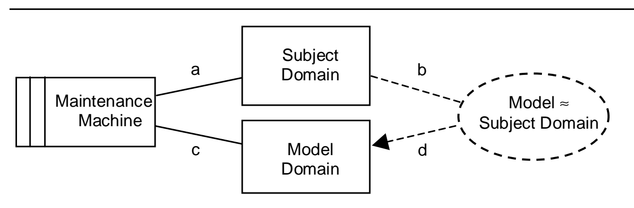

Fig. 2. Maintaining Model-Subject Correspondence

The Model Domain, which is a part of the Machine in the original whole problem, here becomes a component of the Problem Domain.

As the diagram shows, there are four distinct sets of phenomena to be considered in this problem:

- The phenomena between which the correspondence must be maintained:
  - the phenomena b in the Subject Domain; and
  - the phenomena d in the Model Domain.
- The phenomena which the Machine shares with each domain:
  - the phenomena a of the Subject Domain; and
  - the phenomena c of the Model Domain.

For example, in the lift system the phenomena b may be the existence or absence of an outstanding request for the lift from *floor[f]*, and the phenomena d may be the results of a boolean method *Floors.Requesting(f)*. The required correspondence is that *Floors.Requesting(f)* returns *true* exactly when there is an outstanding request[7].

[7] As the arrowhead indicates, the correspondence must be achieved by constraining the Model, not the Subject Domain.

To achieve the correspondence the Machine monitors phenomena a—for example, each pressing of a request button—and causes phenomena c—for example, by invoking a *Floors.ButtonPressed(f)* method. It must also, of course, monitor the lift behaviour as detectable at a, and execute appropriate operations at c in response.

To solve the maintenance subproblem—that is, to devise a Machine that can achieve and maintain the required correspondence—we must, in principle, describe the properties of the Subject and Model Domains. The Subject Domain description will show how the creation and extinction of outstanding requests b are *imperfectly* inferred from the shared phenomena a[8]. The Model Domain description will capture the design of the object model, showing how the results returned by invocations of *Floors.Requesting(f)* are *fully* determined by the preceding invocations of *Floors.ButtonPressed(f)* and of other methods.

Clearly, these two descriptions are not identical. If we mean to deal meticulously with the maintenance subproblem, and to identify clearly the respects in which the Model fails to correspond to the Subject reality, we must not be content with just one description, vaguely characterised as our 'model of the lift'. One description can be true to the Subject Domain, or to the Model Domain, but not to both[9].

It is worth mentioning one respect in which the divergence of the Model from the Subject Domain is particularly liable to cause serious difficulty and even major system failure. The Model and the Subject Domains have, in general, different life spans: the life span of the Model begins when the Machine is started (or, it may be, restarted), but the life span of the Subject Domain, in almost every case, begins earlier. The Model is easily *initialised* because it is essentially a local variable of the Machine, but the Subject Domain can not be treated so imperiously. A horrible example of a failure to understand this point is the design of a GPS military device for calculating target co-ordinates. After a battery change the device is programmed to reinitialise the target co-ordinates to its own location. In one incident three soldiers were killed and 20 injured by friendly fire as a result of this defect[10].

7 Summary

The purpose of this short paper has been to present a few basic tenets of an explicit and careful approach to description (or 'modelling'). They might be summarised as: "Distinguish the machine from the problem domain", "Don't

[8] The inference is imperfect because the shared phenomena are not sufficient, for example, to indicate whether a requesting user has, in fact, had enough time to enter the lift.

[9] One description could serve for both if the Model were, very exceptionally, a perfect surrogate for the Subject Domain. It would then be necessary to provide two mappings from terms in the common description to phenomena in the two domains.

[10] Steve Ferg drew my attention to this incident, reported in the Washington Post of 24 March 2002.

restrict description to the machine", and "State explicitly what is described". These aspects of descriptive technique are far from new (indeed, they have been presented elsewhere[1, 5, 2]). But they are surprisingly often ignored, and the claim of this paper is that they merit more attention than they commonly receive.

8 Acknowledgements

This paper has benefited greatly from many conversations with Roel Wieringa, and has been improved by his comments and suggestions in response to an earlier draft. It has also been improved by some suggestions of the editor of SoSym.

References

1. Michael Jackson; Requirements and Specifications: a Lexicon of Practice, Principles and Prejudices; Addison-Wesley, 1995
2. Michael Jackson; Problem Frames: Analysing and Structuring Software Development Problems; Addison-Wesley, 2000
3. Philippe Kruchten; The Rational Unified Process: An Introduction; Addison-Wesley, 1999
4. John von Neumann and Oskar Morgenstern; Theory of Games and Economic Behaviour; Princeton University Press, 1944
5. Pamela Zave and Michael Jackson; Four Dark Corners of Requirements Engineering; ACM Transactions on Software Methodology 6,1, July 1997, 1–30