'Michael Jackson's best work ever.' Tom DeMarco

Software Requirements & Specifications

a lexicon of practice, principles and prejudices

MICHAEL JACKSON

acm PRESS

ADDISON-WESLEY

INTRODUCTION

Software development has many aspects. David Gries sees it as logic and mathematics. Enid Mumford sees it as a socio-ethical challenge. Kristen Nygaard sees it as a field of labour employment relations. Watts Humphrey sees it as a problem in manufacturing process control. Other people see it as a kind of engineering – an activity of building useful things to serve recognizable purposes. I'm one of those people. The engineering aspects interest me most. So of course I claim that they are the most important.

The Machine and the World

To develop software is to build a MACHINE, simply by describing it. This is one good reason for regarding software development as engineering. At a general level, the relationship between a software system and its environment is clear. The system is a machine introduced into the world to have some effect there. The parts of the world that will affect the machine and will be affected by it are its APPLICATION DOMAIN. That's where your customers will look to see whether the development has fulfilled its intended purpose. You don't gauge the success of a theatre reservations system by looking at its computers. You look at the world outside. Can people book seats easily? Are the theatres full? How long does it take to pick up a ticket? Are all the credit card payments properly accounted for? Are cancellations handled smoothly?

This distinction between the machine and the application domain is the key to the much-cited (but little understood) distinction between WHAT AND HOW: *what* the system does is to be sought in the application domain, while *how* it does it is to be sought in the machine itself. The problem is in the application domain. The machine is the solution.

Focusing on Problems

An inability to discuss problems explicitly has been one of the most glaring deficiencies of software practice and theory. Again and again writers on development method claim to offer an analysis of a problem when in fact they offer only an outline of a solution, leaving the problem unexplored and unexplained. Their readers must work out their own answers to the

question: If this is the solution, what was the problem? Ralph Johnson, a leading advocate of design patterns in object-oriented development, has this to say:

> 'We have a tendency to focus on the solution, in large part because it is easier to notice a pattern in the systems that we build than it is to see the pattern in the problems we are solving that lead to the patterns in our solutions to them.'

That's unfortunately true. And this tendency to focus on the solution has been harmful both to individual development projects and to the evolution of methods. Many projects have failed because their REQUIREMENTS were inadequately explored and described. The requirements are located in the application domain, where the problem is; but in most developments all the serious documentation and discussion focuses on the machine which is offered as a solution to the problem. At best there will be a careful description of the coastline where the machine meets the application domain. But the hinterland of the application domain is too often left unexplored and unmapped.

Domain Descriptions

Paying serious attention to the application domain means writing serious, explicit and precise DESCRIPTIONS. Don't think it's good enough to interview a few people and make a few rough notes. Don't say: 'Everyone knows what you do when you book a theatre seat', or 'Everyone knows that the seats become available when the show is scheduled'. What everyone knows is often wrong, and always very far from complete.

Above all, don't think it's good enough to describe the processing that your system will do. If you think of your system as processing input and output flows of data, then the application domain you must describe is what's at the other end of those flows. If you think of your system as processing transactions, or operations, or use-cases, then the application domain you have to describe is the world where the transactions or operations or use-cases are initiated. The processing of the data flows, or transactions, or operations, or use-cases is what the machine does. It's part of the solution, not part of the problem. Leave it till later.

Modelling

There's a big temptation to believe that you can describe the application domain and the machine all together, in one combined description. The justification for this belief is that some part of the machine is often a MODEL of some part of the application domain. That means that there's some

description that's true both of the machine and of the application domain. So why not just write that description and save some duplication of effort?

Here's why. A description of a modelling relationship is always incomplete: there's more to say about the application domain, and more to say about the machine too. You need to say those things somewhere. You ought not to say them in the common description, because they're not common. In principle you really need three descriptions: the common description; the description that's true only of the machine; and the description that's true only of the application domain. If you make all of those descriptions, and separate them carefully, you'll be all right.

But if you make only one description, you will surely be tempted to put things into it that describe only the machine, and to leave out things that describe only the application domain. After all, you have to describe the machine sooner or later, don't you?

You can see the results clearly in many object-oriented modelling descriptions. Often they are accompanied by fine words about modelling the real world. But when you look closely you see that they are really descriptions of programming objects, pure and simple. Any similarity to real-world objects, living or dead, is purely coincidental.

Description Technique

Description is at the heart of software development. It is not an exaggeration to say that the whole business of software development is making descriptions. Programs are descriptions of machines; requirements are descriptions of the application domain and the problems to be solved there; and specifications are descriptions of the interface between the machine and the application domain. Every product of development is a description of some part or aspect or property of the problem or of the machine or of the application domain.

That's why the techniques of description are of prime importance for developers. The importance of some issues of descriptive technique is well recognized: language is obviously important, because language is the RAW MATERIAL of description. But some other aspects have not been given their due prominence:

- DEFINITION. You must distinguish between defining new terms and using existing terms to make statements. Without this distinction it's impossible to know whether you're talking about the world or talking about the terminology for talking about the world. Yet it's a distinction that's often ignored or blurred.
- Description SCOPE and SPAN. The scope of a description restricts the *classes of phenomena* it can talk about. The scope of a world rainfall map is different from the scope of a world population map. The span

restricts the *area of the world* it can talk about. The span of a rainfall map of France is different from the span of a rainfall map of Italy. If you choose the wrong scope or span for a description you've shot yourself in the foot. You'll find your work can only limp along.

- REFUTABLE DESCRIPTIONS. Most – perhaps all – of your descriptions should be refutable. A refutable description says something precise about the world that could be refuted by a counter-example – for example: 'No company employee works on more than one project at any one time'. Find an employee working on two projects at one time and you have shown conclusively that the description is not true. Descriptions that can't be refuted don't say much.
- PARTIAL DESCRIPTION. Because the application domain is usually large and complex, you need ways of *separating concerns*. That just means not thinking about everything at once, but structuring a complex topic as a number of simpler topics that you can consider separately. Structure is even more important in analysis than in program design.
- HIERARCHICAL STRUCTURE is one way of separating concerns. Usually it doesn't work well and some kind of parallel structure is better. The separation of a coloured picture into cyan, magenta, yellow, and black overlays is a better metaphor for description structuring than the hierarchical assembly structure of parts in manufacturing. Hierarchical structure is the sand on which TOP-DOWN methods are built.
- It's important to separate descriptions of different MOODS. A description in the *optative* mood says what you want the system to achieve: 'Better seats should be allocated before worse seats at the same price'. A description in *indicative* mood says how things are regardless of the system's behaviour: 'Each seat is located in one, and only one, theatre'.

Methods and Problem Frames

Failure to focus on problems has harmed many projects. But it has caused even more harm to the evolution of development METHOD. Because we don't talk about problems we don't analyse them or classify them. So we slip into the childish belief that there can be universal development methods, suitable for solving all development problems. We expect methods to be panaceas – medicines that cure all diseases. This cannot be. It's a good rule of thumb that the value of a method is inversely proportional to its generality. A method for solving all problems can give you very little help with any particular problem.

But have you ever read a book on a development method that said: 'This method is only good for problems of *class X*'? Probably not. It's not

surprising, because without an established discipline of analysing and classifying problems there can't be a usable vocabulary of *problem classes X*, and *Y*, and *Z*.

Classifying problems and relating them to methods is a central theme of this book. The crucial idea here is the idea of a PROBLEM FRAME, derived from the work of POLYA. A problem frame defines a problem class, by providing a ready-made structure of *principal parts* into which all problems of the class must fit. Whether a particular problem fits a particular frame depends on the structure and characteristics of the application domain, and the structure and characteristics of the requirement.

A good method addresses only those problems that fit into a particular problem frame. It exploits the properties of the frame and its principal parts to give systematic and sharply focused help in reaching a solution. This means that the power of a method depends on the quality and precision of the frame. There's a principle here:

- *The Principle of Close-Fitting Frames*
  The purpose of a problem frame is to give you a good grip on the problem. So the frame must define and constrain the problem very closely indeed.

You can't work effectively on a problem that's wobbling around in a sloppy, loose-fitting frame.

Problem Complexity

The largest description structures in software development are the structures that arise from PROBLEM COMPLEXITY. A complex problem is a problem that cannot be completely accommodated by any available frame and completely solved by any available method. You need more than one view of the same problem. That's not just taking more than one view of the machine – for instance, data, function, and state transition, or class hierarchy, object aggregation, and object interaction. It's more like recognizing that the same problem is simultaneously a control system problem, an information system problem, and a message switching problem. It's a MULTI-FRAME PROBLEM.

The task, then, is to separate out the different problem frames in their parallel structure. Identify the principal parts of each frame; and identify the overlapping parts and aspects of the application domain and requirement that they cover. This is the essence of problem decomposition. When you separate a problem by problem frames, you're separating it into simpler sub-problems *that you know how to solve*. Then you have a real chance of mastering the complexity that is found in every realistic software development.