Some Complexities in Computer-Based Systems and Their Implications for System Development

Michael Jackson
101 Hamilton Terrace
London NW8 9QX
+44 71 286 1814
+44 71 266 2645 (fax)

Abstract

Mastering complexity is central to the software development task, and separation of concerns is our chief tool. An informal framework is proposed within which concerns may be identified. An example of a completely simple system is given, and some sources and cases of complexity are considered with respect to the framework. In the final section some implications for software development method are discussed.

1 Introduction

The subject of this paper is complexity. By this I do not mean computational complexity, but rather structural complexity, the complexity that manifests itself as difficulty in developing a system and achieving confidence in its correctness. This kind of complexity is not absolute, but relative to our ability to identify, understand and overcome the difficulties that arise: we hope that what is complex today will be simple tomorrow. There is some basis for such a hope in the experience of certain specialised applications: compiler writers now, for example, routinely produce compilers for languages that thirty years ago would have been thought impossibly difficult to compile.

In one sense, to talk convincingly about structural complexity is either impossible or self-defeating. If a complexity is intelligibly explained, it becomes ipso facto no longer a complexity. If it is not intelligibly explained, then nothing convincing has been said. I shall rely on this observation in hoping for your indulgence if much of what I am going to say seems rather vague and imprecise.

I think we all have an intuitive idea of what we mean by complexity in the sense I am considering it. Complexity is the kind of difficulty that can be surmounted by a well chosen separation of concerns, to use Dijkstra's phrase. Faced by a complex problem, we tackle it by separating it into a number of concerns, each of which is simple: simple because we can grasp it intellectually without further separation.

But separation alone is not enough. Having started with one problem, we must end with one solution. So the concerns we have separated must be recombined: having divided to conquer, we must then reunite to rule. The compiler writer who has recognised that the problem can be separated into lexical analysis, syntactic analysis, code generation, and code optimisation, must find a way of combining the solutions to these four subproblems into a single program.

So I want to discuss complexity in terms of separation and composition of concerns. But 'concerns' is too vague a term, and we can be at least a little more exact if we talk of the descriptions, formal and informal, textual and diagrammatic, in which a developer captures concerns. We may regard software development as an activity in which descriptions are created, manipulated, and composed, necessarily including the creation of a set of descriptions that can be interpreted by a machine as an executable program. Complexity, then, is a relationship among descriptions; a complex development problem is one for which it is difficult to separate, create and compose the necessary descriptions.

To talk about relationships among descriptions we must have a conceptual framework within which the descriptions can be placed. In a paper discussing system complexity, Winograd[1] suggests that 'we need to shift our attention away from the detailed specification of algorithms, towards the description of properties ...'. He offers a framework of three domains of description: subject, interaction, and implementation.

In the main part of this paper I shall propose a somewhat more elaborate framework. It is inevitably rather imprecise, but I hope it will be clear enough to support the discussion that is built on it. Section 2 introduces this framework by describing a completely simple system development: just as we need to understand sequentiality in order to understand concurrency, so we need to understand simplicity in order to understand complexity. Section 3 discusses what it is that makes this system so simple. Section 4 gives an account of various forms and sources of complexity. Section 5 draws some implications for system development methodology.

2 Joe's Dance Hall

2.1 The Problem

Joe runs a dance hall. When the hall is open for a dance, people arrive in groups throughout the evening. Some people leave before the dance ends, not necessarily in the groups in which they arrived. There is no band - just an automated disco - so the dancers are the only occupants.

Joe sits at his cash desk. On one side there is a turnstile to admit people to the hall; on the other there is a passage by which people can leave. Both the turnstile and the passage are under Joe's close observation as he sits at his cash desk. Joe needs a system that will ensure that he obeys the Fire Department's restriction that no more than 100 people may be in the hall at any one time. When he has to refuse admission, he wants to display an apologetic sign that says 'Sorry, We're Full!'.

2.2 Joe's System Development: the View

Joe needs to clarify his understanding of the problem. Wisely, he starts by making a description of the Subject Domain - the real world about which the system will compute. He writes down:

Subject Domain:

```
  arrivals = arrive-group*;
  departures = depart-group*;
```

He then proceeds to describe the Target Domain of the system - the real world that it will control:

Target Domain:

```
  admissions = admit-group*;
  apologies = apologetic-sign*;
```

Now he must describe the Function. This is not necessarily a function in the mathematical sense, but any description that specifies the control that the system will exercise over the Target in accordance with the properties - in this case, the time-varying properties - of the Subject:

Function:

```
  each arrive-group
        either becomes an admit-group
            or receives an apologetic-sign;
  occupancy-at-time-t =
      sum-at-time-t(admit-groups) -
      sum-at-time-t(depart-groups);
  for any time t:
      occupancy-at-time-t <= FD-Limit;
```

He still needs to describe the User Domain - the agency that will determine when the system must start and stop computing. He will start computing at the beginning of the evening and stop at the end:

User Domain:

```
  usage = start stop;
```

Finally, he must describe the Machine Domain:

Machine Domain:

```
  my PC with MS-DOS and Pascal including the
  special routines.
```

Joe's PC has a Pascal compiler with two built-in library routines that can operate the turnstile and briefly illuminate the apologetic sign through one of the ports.

Joe has now completed, albeit somewhat informally, what I will call the View: a description of the system from the outside, that we might refer to as a requirement or perhaps a specification. The View shows how the system fits into the world.

2.3 Joe's System Development: Implementation

Joe now needs to realise or implement his system. He needs to describe an Implementation.

First, he describes the Argument Interface - the interface through which the system will receive inputs that keep it informed about the Subject Domain. He decides that he will enter an integer at the keyboard for each arrive-group and for each depart-group, positive for arrival, negative for departure:

Argument Interface:

```
  events = (number-arriving|
            minus-number-departing)*;
  number-arriving: integer;
  minus-number-departing: negative integer;
```

Next, the Result Interface - the interface through which the system will control the Target:

Result Interface:

```
  outputs = (apology|admit-number)*;
```

The Interaction Interface is where execution is started or stopped, or perhaps suspended or resumed. Joe likes to keep it simple, and he doesn't need any suspension or resumption. So he plans to start computation by typing 'Dance', and stop it by typing Control-C:

Interaction Interface:

```
  controls = 'Dance' Control-C;
```

There is no point in giving a detailed description of the fourth Interface, which is the Execution Interface: this is already described in his Pascal manual. But he reminds himself of the forms of procedure call to operate the turnstile and apologetic sign:

Execution Interface:

```
  program-language =
     Pascal with procedure admit(n:integer);
           and procedure apologise;
```

Finally he must construct the Program itself, which is the fifth part of the Implementation. To do this, he must compose the four Interface descriptions, satisfying the Computation description to ensure that the system provides the correct interleaving of behaviours and values at the Interfaces. He recognises that his description of the Argument Interface is not quite enough for this purpose, and he improves it to:

Argument Interface:

```
  events = (number-arriving|
          minus-number-departing)*;
  number-arriving: integer;
  minus-number-departing: negative integer;
  number-arriving = (OK|not-OK);
```

The distinction between OK and not-OK is, of course, necessary because of the Computation rules: not all arrive-groups can be admitted. After a little more manipulation of the descriptions, and some composition work, he produces the Program. Here it is:

```
  PROGRAM dance(input);
   CONST
    fdlimit = 100;
   VAR
    occupancy:integer;
    event:integer;
   BEGIN
    occupancy:=0;
    WHILE true DO
     BEGIN
      readln(event);
      IF event < 0 THEN
       occupancy:=occupancy+event
      ELSE
       IF (occupancy+event) > fdlimit THEN
        apologise
       ELSE
        BEGIN
         admit(event);
         occupancy:=occupancy+event
        END
     END
   END.
```

Joe has now completed his View and his Implementation, and is in good shape. We may criticise him for his informality, but we could help him with that, I am sure. Also, the system is not exactly sophisticated, and we could help him to improve it in many ways. But methodologically his work has been very sound, and he merits our approval.

3 What Makes Joe's System So Simple

Joe's methodological framework has no claims to universality. Many arguments could be made against it. For example, it could be argued that the distinction between Subject and User Domains, and the analogous distinction between Argument and Interaction Interfaces, is artificial and difficult to sustain. Certainly, users of SADT[2], where an actigram box has four interfaces corresponding broadly to the Argument (input), Interaction (control), Result (output), and Execution (mechanism) Interfaces, often complain that the distinction between input and control is mysterious. It could be argued that information systems do not have a Target Domain, and that in embedded systems there is no distinction between Subject and Target Domains. Further, in many common implementations, such as those that use a procedure call interface, the Argument, Result, and Interaction Interfaces are implemented together in a single interface.

There is force in these arguments, but I hope that we shall see that the framework is nonetheless useful. It is important to remember that our purpose here is to use the framework not as a prescriptive framework of a method, but as a structure within which we can explore relationships among descriptions and hence among concerns. More difficulties arise from insufficient than from excessive separation of concerns.

3.1 Single View and Implementation

One reason that Joe's system is simple is that he needs only one View and one Implementation, simply related. The Argument Interface in the Implementation provides the information needed for the system to model the Subject Domain; the Result Interface transmits the outputs that control the Target Domain; the Interaction Interface transmits the commands from the User Domain; the Execution Interface runs directly on the Machine Domain. It is trivially easy to see how and where each part of the View is implemented.

Joe's problem exhibits the simplest pattern of relationships between Views and Implementations. Purely for mnemonic purposes we might depict it as:

```
 ----------------           --------------------
| View:          |<------->| Implementation:    |
|                |         |                    |
|    Subject <---------------> Argument         |
|                |         |                    |
|     Target <---------------> Result           |
|                |         |                    |
|       User <---------------> Interaction      |
|                |         |                    |
|    Machine <---------------> Execution        |
|                |         |                    |
|   Function <---------------> Program          |
|                |         |                    |
 ----------------           --------------------
```

3.2 Easy Separations

Joe had little trouble in separating the Domains in his View. In particular, the Target Domain is quite separate from the Subject Domain, so he does not have to worry about interactions between them. He achieved a modest insight in recognising that the admit-groups are in the Target, while it is the arrive-groups that are in the Subject Domain.

3.3 Simple Descriptions

Joe's problem admits of simple descriptions of each Domain and Interface, and of the Function. Although Joe is not much of a formalist, his descriptions are reasonably intelligible and do not resort much (except, perhaps, in his Function description) to hand-waving. For the most part, he gets by with regular expressions, which quite adequately describe the Subject and Target Domains and map easily to the Argument and Result Interfaces. If his problem had been different, he might have used recursive functions or predicate logic for his descriptions. This would not be any harder than using regular expressions, provided that he chooses his language well for its purpose; but he might well have found difficulties if he had been forced to use two or more fundamentally different languages in combination.

In all systems that deal with the natural world there is a need to formalise what is inherently informal. Joe has formalised his Subject Domain and a part of his Target Domain in terms of groups of people, and in the Function he has described occupancy in terms of arithmetic about the admissions and departures. Of course, this formalisation - like all formalisations of the informal - is imperfect: if a pregnant woman is admitted to the hall and delivers her baby before departing, Joe's occupancy figure will be wrong. But if the Fire Department complains he will rely on the standard justification: his accuracy is close enough for government work.

3.4 Easy Composition

The Function, of course, refers to both the Subject and Target Domains because it describes the relationships that the system must maintain between them. The Program, by contrast, must not only refer to the Argument and Result Interfaces: it must actually be formed by a composition of those interfaces, along with the Interaction and Execution Interfaces also. Joe's task was easy here too.

3.5 The Nature of Composition

What I mean by composition is the opposite of abstraction. If we are given a program, we may ask a question such as 'what is the grammar of the input to this program?', and we can answer the question by making an abstraction - an incomplete description - of the program. This incomplete description will pay attention only to the way in which the program traverses and analyses its input, and hence will show the structure or grammar that it imposes on that input. Certain such abstractions are called 'slices'[3], and are often used intuitively in reading and understanding programs.

Composition is the opposite of abstraction in the sense that we start with several given descriptions and form another description such that each of the original descriptions is an abstraction of it. The permissible compositions are constrained by the relationships that must hold among parts of the original descriptions when they are composed.

Given his Interface descriptions, Joe had to make a Program such that each Interface description is an abstraction of the Program, and that the Interfaces are correctly related. For example, he had to satisfy the part of the Function description that specified that each arrive-group either becomes an admit-group or receives an apologetic sign.

Essentially, Joe was composing descriptions that are all regular expressions. Because there is no structure clash[4,5], Joe was able to achieve his composition easily. He would have found it a lot harder if his descriptions had been written in different languages, or if there had been a structure clash between two descriptions.

3.6 Composing the Interaction and Execution Interfaces

The idea of composing the Interaction and Execution Interfaces with the other two may seem surprising. In fact, it is always necessary, but often, as in this case, trivial.

The Interaction is a sequence of the MS-DOS command to run the program, followed by Control-C, with the rest of the required behaviour bracketed by these. The MS-DOS Pascal system provides this composition automatically: it would not have done so if Joe had decided to implement 'stop' by entry of a special integer value such as zero or 9999.

The Execution Interface places few constraints on the Program other than those that the compiler will check. There is no ordained pattern of invocation of the special procedures - they can be invoked at any time and in any order. The only significant constraint is that Pascal variables, unlike those in some COBOL systems, must be initialised before they are used; this constraint is ubiquitous and its satisfaction hardly merits explicit consideration. But in principle Joe must compose an instance of the description

```
  variable = assignment(assignment|use)*
```

into his Program structure for each variable he needs. He would have had to work harder if he had been using a library of abstract data types with significant constraints on the order of operations for each type.

4 Complexities

In this section I shall discuss some sources and cases of complexity in terms of relationships among concerns as reflected in descriptions: that is, among Views and Implementations, Domains and Interfaces, Functions and Programs. In most of the cases I shall use diagrams to depict relationships: not because these diagrams are formal manipulable descriptions (they are not), but because I think they have a short-term mnemonic value: although they are completely informal, they help to hold in mind the relationship that is being discussed.

The order in which the complexities are presented is not intended necessarily to reflect an ordering in their severity, but rather to make them easier to explain and understand. However, I shall start with two complexities that are so well known and understood as scarcely to count as complexities at all.

4.1 Horizontal Separation of Function

For the classic problem of lexical and syntactic analysis there is a standard solution: separate the lexical from the syntactic concerns, and define the syntactic analysis over the domain of the lexemes produced by the lexical analysis. This is a very neat and effective separation. If we try to use one View instead of two, our Subject (the input strings) will demand an elaborate description to cover both lexical and syntactic properties, and the Function description will also be elaborate.

Within the framework of this paper, I would explain this as a separation into two Views: the stream of lexemes constitutes the Target Domain of the first View and the Subject Domain of the second View. We may depict the relationship between the Views as:

```
 -------------------           --------------------
| View1:            |         | View2:             |
|                   |         |                    |
|   Subject1        |     ------> S2(lexemes)      |
|                   |    |    |                    |
|   T1(lexemes) ---------     |   Target2          |
|                   |         |                    |
|   User1           |         |   User2            |
|                   |         |                    |
|   Machine1        |         |   Machine2         |
|                   |         |                    |
|   Function1       |         |   Function2        |
|                   |         |                    |
 -------------------           --------------------
```

This separation, which is applicable to many problems, exacts a price: we now have two Implementations. This is not in itself a difficulty, except for the fact that we must compose them to give a single Interaction and a single Execution Interface. The simplest composition is to connect two programs by a serial file and provide a JCL procedure at the Execution Interface that invokes them in sequence. But this solution may be unacceptably inefficient, and we may seek others: since Conway's original paper[6] and the introduction of coroutines by Simula[7], many solutions in terms of language, execution environment, and manipulation of program texts have been widely known and used.

The most popular and effective solutions to the composition problem are, rightly, those that demand the least manipulation of the descriptions to be composed. We aim for a 'modular structure' in implementation that mirrors the 'modular structure' of our specification. A wise aim, but often quite beyond our reach. Parnas' seminal paper on modularisation[9] is widely cited, but few remember his observation that the specification modules must often be reassembled into a different configuration for an efficient implementation.

4.2 Vertical Separation by Machine

The problem faced by the developer of an operating system on a bare hardware machine is my second classic complexity. One very well-known solution[8] is to construct a layered hierarchy of machines, with the bare hardware at the bottom layer and the complete operating environment at the top, each layer depending on the layer below it and supporting the layer above.

We may then depict the relationship between two adjacent layers as:

```
 -------------------           --------------------
| View1:            |----     | View2:             |
|                   |    |    |                    |
|   Subject1        |    |    |   Subject2         |
|                   |    |    |                    |
|   Target1         |    |    |   Target2          |
|                   |    |    |                    |
|   User1           |    |    |   User2            |
|                   |    |    |                    |
|   Machine1        |     ------> Machine2         |
|                   |         |                    |
|   Function1       |         |   Function2        |
|                   |         |                    |
 -------------------           --------------------
```

View1 provides the Machine Domain for View2; Implementation1 defines the Execution Interface for Implementation2. The Argument, Result, and Interaction Interfaces of Implementation1 are combined, and form the other side of this interface.

There are many familiar ways of implementing this scheme, including the use of procedure calls, perhaps to instances of abstract data types. And, of course, an additional layer may be interposed in the form of a new 'high-level' language with its associated compiler or interpreter.

We could regard Joe's special procedures as an example of a solution to a problem of this kind. A trivial View of the turnstile and a trivial View of the apologetic sign have been combined with standard Pascal to give the Machine Domain of his system.

4.3 Inseparability of Subject and Target

Separation of Subject and Target Domains provides an aspect of simplicity that we are perhaps inclined to value only when it is impossible. The Subject Domain is outside our control: we must compute about it, but we can not affect it; the Target Domain is within our control, but we need not consider how, if at all, the changes in the Target Domain affect the Subject Domain. On-the-fly garbage collection does not allow this separation: this is one reason why it is difficult to design and program.

Another example, of a different kind, is the separation of syntactic and semantic functions in natural language processing, and the similar, though much easier, problems that arise when we try to read ill-formed handwriting in a letter from a friend. In this second case we would like to make the separation analogous to the compiler writer's separation of lexical and syntactic analysis, but this time separating character and word analysis. Unfortunately, we may often find that characters can be recognised only by considering the word that would be formed:

```
    -------------------           --------------------
   | View1:            |         | View2:             |
   |                   |         |                    |
   |   S1(squiggles) <-----   -----> S2(chars)        |
   |                   |   \ /   |                    |
   |                   |    X    |                    |
   |                   |   / \   |                    |
   |   T1(chars) ----------   ------ T2(words)        |
   |                   |         |                    |
   |   ...             |         |   ...              |
   |                   |         |                    |
    -------------------           --------------------
```

We can not describe the Subject1 Domain of squiggles adequately for character recognition unless we include some description of the resulting words.

4.4 Reporting Execution Behaviour

Suppose that we are planning a new version of a hardware machine or of a compiler or interpreter, intended to make programs run faster than the old version. To help us to direct our efforts appropriately, we want to collect information about relative usage of different instructions. We have a system2 that is run regularly on the old version, and we are prepared to regard it as typical; we need an information system1 whose Subject is the execution behaviour of system2. Static analysis of the programs and data of system2 is, unfortunately, impossible.

```
    -------------------           --------------------
   | View1:            |         | View2:             |
   |                   |         |                    |
   |   Subject1 <------------    |   Subject2         |
   |                   |     |   |                    |
   |   ...             |     |   |   ...              |
   |                   |     |   |                    |
    -------------------      |    --------------------
                             |   | Implementation2:   |
                             |   |                    |
                              -----  Execution2       |
                                 |                    |
                                 |   ...              |
                                 |                    |
                                  --------------------
```

The two Views are well separated, and the problem arises mostly in the collection of information about Subject1. If we are working on a new interpreter, we could embed operations in the old interpreter to provide the necessary information. If we are working on a new compiler, we would probably want to post-process the program texts of system2 to embed those operations there. If we are working on a new hardware machine we would not want to modify the old one, so we might attach some kind of sensors to the hardware or perhaps use a sampling technique involving a software package that interrupts execution and logs the instruction found at the current value of the execution counter.

4.5 Scheduling Process Execution

A more substantial complexity may arise in scheduling execution of processes that communicate by buffered message streams. Suppose that within a single given development problem we find it necessary to develop a Program that is a composition of this kind: separation of concerns has dictated that we separate the data flow graph and the content of the flows, on the one hand, from the scheduling of the processes on the other hand. Now the time has come to compose the result with at least a modicum of scheduling control: we want to ensure that process X does not progress past a certain point until process Y has begun execution, or that process W waits at a certain point until all processes in the set Z have reached the end of their executions.

If the processes are in the Implementation of View2, and the scheduling problem is regarded as View1, we have:

```
 ------------------            --------------------
| View1:            |         | View2:             |
|                   |         |                    |
|   Subject1 <------------    |   Subject2         |
|                   |     |   |                    |
|   Target1 <-------------|   |   ...              |
|                   |     |   |                    |
|   ...             |     |    --------------------
|                   |     |   | Implementation2:   |
 ------------------       |   |                    |
                           -----  Execution2       |
                              |                    |
                              |   ...              |
                              |                    |
                               --------------------
```

The Execution Interface of Implementation2 is both the Subject and the Target of View1. As in Reporting Execution Behaviour in 4.4 above, the difficulty lies more in finding a satisfactory implementation of the composition of View1 and View2 than in making the separation: it is not difficult to specify View1. The difficulty is compounded if the mechanisms for process activation and suspension lie entirely within the machine, as they probably will:

```
 -------------------           --------------------
| View1:            |         | View2:             |
|                   |         |                    |
|   Subject1 <------------    |   Subject2         |
|                   |     |   |                    |
|   Target1 <-------------|   |   ...              |
|                   |     |   |                    |
|   ...             |     |-----  Machine2         |
|                   |     |   |                    |
 -------------------      |    --------------------
                          |   | Implementation2:   |
                          |   |                    |
                           -----  Execution2       |
                              |                    |
                              |   ...              |
                              |                    |
                               --------------------
```

Now we will have a significantly more complex composition to do: we must deal with Machine2 as a Subject and a Target in its own right, in addition to dealing similarly with the Execution2 Interface between Machine2 and the Program that runs on it.

4.6 Data Processing Input-Output Subsystem

A separation of concerns that is desirable in many data processing systems is the separation between the substantive system, computing about the organisation's affairs, and what has sometimes been called the 'input-output subsystem'. This subsystem is concerned with such matters as the grouping of input and output operations into transactions, deciding which transaction types are permitted at different terminals, providing protocols within which operators can move conveniently from one class of transaction to another that is likely to be needed next, and so on.

If the substantive system is View2 and the input-output subsystem is View1, we can depict their relationship as:

```
 -------------------           --------------------
| View1:            |         | View2:             |
|                   |         |                    |
|   Subject1 <------------    |   ...              |
|                   |     |   |                    |
|   Target1 <-------------|    --------------------
|                   |     |   | Implementation2:   |
|   ...             |     |   |                    |
|                   |     |-----  Argument2        |
 -------------------      |   |                    |
                          |-----  Result2          |
                          |   |                    |
                           -----  Interaction2     |
                              |                    |
                              |   ...              |
                              |                    |
                               --------------------
```

The Argument, Result, and Interaction Interfaces of the substantive system, taken together, all form both the Subject and the Target of the input-output subsystem. Notice that Subject1 and Target1 descriptions will be abstractions of the Interfaces they describe: they will be concerned with input and output data classes, but not with the content that distinguishes instances of those classes.

4.7 Dynamic Network Reconfiguration

A certain radio communication system consists of a number of nodes of different types that can move freely from place to place; they communicate over the arcs, which are provided by channels of different radio frequencies. Because the communication needs of the users are continually changing, because it may be necessary to switch frequencies for security reasons, and because in certain circumstances nodes may leave the network and new nodes may join, it is necessary to reconfigure the network dynamically while it is in operation. The reconfiguration commands themselves must be sent to the appropriate nodes, using the network itself to send the commands.

Ignoring Implementations altogether, the relationship between View1 which is concerned with the substantive use of the network and View2 which is concerned with its reconfiguration is:

```
 -------------------           --------------------
| View1:            |         | View2:             |
|                   |         |                    |
|   Subject1        |      -----> Subject2         |
|                   |     |   |                    |
|   Target1         |     |-----> Target2          |
|                   |     |   |                    |
|   User1           |     |   |   User2            |
|                   |     |   |                    |
|   Machine1 <------------------> Machine2         |
|                   |         |                    |
|   Function1       |         |   Function2        |
|                   |         |                    |
 -------------------           --------------------
```

The network is, in effect, a machine. It is the machine for both View1 and View2. It is also both the Subject and the Target for View2. View2 is monitoring and changing the machine on which Implementation2 runs, while Implementation2 is in course of execution; and it is also monitoring and changing the machine on which its own Implementation runs. The developers of this system will certainly need to pay careful attention to making several different but consistent abstractions of this machine, and to the Execution Interfaces it supports.

5 Some Implications for System Development

We may regard system development as the activity of making, manipulating, and composing descriptions for some purpose; the purpose is achieved when we have produced a description of a computation appropriate for the purpose and capable of execution by available machinery, together with whatever other associated descriptions are demanded by the customer, users, maintainers, and anyone else with a legitimate interest in the system.

The central questions in planning and managing a development are: what descriptions must we make? in what languages? by what operations? and in what order? These, of course, are the questions that a development method purports to answer.

5.1 How Many Descriptions Must We Make?

The more descriptions we make, the more effort will be needed for technical and managerial control of the development. There is a significant cost associated with having many pieces of paper in our filing cabinet or many disk records in our development support machine. And we will also have a lot of compositions to do, putting descriptions together. So it is always tempting to reduce the number of descriptions just as far as we believe our abilities will allow. Hackers are people with a high confidence in their abilities in this respect.

If we make too few descriptions, they will be difficult to make and difficult to understand, so development will be much more error-prone. We won't avoid the compositions - we will just be doing them secretly and implicitly in our heads instead of explicitly and openly on paper or on keyboard and screen. Many of the descriptions that we or others need to understand the system will be missing: we will be forced to deduce them - if we can - from the descriptions that we do have.

It seems to me that even those of us who are not hackers are inclined to make too few rather than too many descriptions. I think this is because we are heirs to a tradition that began with a single description, in machine language, of each computation, and progressed by devising 'higher-level' languages in which it was possible to make a single description of a more elaborate computation. When the need for specification and requirement descriptions became apparent, 'higher-level' languages were devised for those purposes too, intended to allow essentially single descriptions of specifications and requirements. As a consequence, most of our programming, requirements, and specification languages and associated processors lack all but the most rudimentary mechanisms for composition.

5.2 What Descriptions?

In principle we need to describe the five components of each View and the five components of each Implementation, and also to describe any non-standard relationship among the descriptions. In practice, some of the descriptions will be unnecessary because their topic is trivial or even absent: a pure information system with no element of control has a Result Interface but no Target Domain; Joe's Interaction Interface was so trivial that he could have been forgiven for omitting its description. Sometimes a description is already available before we begin: if the system is to run on an IBM 3090 under MVS/XA with JES3 and CICS and IMS, then the Machine Domain description need be no more than these few morsels from the IBM alphabet soup. But the example of the Dynamic Network Reconfiguration in 4.7 above suggests that we wil sometimes, at least, need to describe our Machine Domain very carefully indeed.

5.3 In What Languages?

We must give the clearest possible descriptions, and the language of each description must be chosen accordingly.

In one particular context, this will lead us to choose natural language for at least a part of a description. To describe an informal Domain we must give a formal description, but we must also say how the informal domain is to be mapped to the formal description, and this can not be done in any formal language. Joe slipped up a little here: the mapping of his Subject Domain to arrive-groups and depart-groups was left implicit in the names chosen; he should have told us more about how we would recognise an arrive-group or a depart-group if we were standing in the foyer of his dance hall.

The most important point about the choice among formal languages is that adequate expressive power in the technical sense does not imply clarity in a particular use. We would not wish to describe a grammar by operations on a state with pre-conditions and post-conditions; we would not wish to describe a finite-state machine in Horn clauses. In a sense, we want to choose the language that allows us to say everything we need while coming closest to allowing us to say nothing else at all.

We must also remember, when we are making descriptions of Interfaces, that we are usually in the realm of realisation, of real computations: to produce the output:

"B is after A"

is not at all the same thing as producing A before B. This is why languages that abstract from execution sequencing can never be enough.

5.4 By What Operations?

The primary operation is creation: describing a View Domain that is outside the system. Creation introduces and describes completely new topics of discourse. Other operations may also introduce new information, but essentially always in relation to existing descriptions. For example, Joe made his Argument Interface by renaming the arrive-groups and depart-groups, choosing to represent them by integers, restating that they are arbitrarily interleaved, and distinguishing OK from not-OK arrive-groups.

The most significant operations for our purpose here are those we use to achieve composition. Because we have traditionally paid too little attention to composition, our intellectual and other tools are at their weakest here: in trying to work with inadequate tools we often introduce gratuitous complexity, that is not in the problem but only in our solution.

Composition of two descriptions requires that, like a dressmaker composing the outer fabric of a garment with the lining, we mark the places at which the descriptions correspond, and then join them at those places. Unlike the dressmaker, we sometimes have the possibility of leaving the final composition to execution time: if we have an executable CSP[10] environment, we may take two process descriptions; rename one of the two alphabets so that the required events correspond; and then simply write the CSP parallel combinator to specify the composition, which will then take effect when the program is executed. More often, we are forced for efficiency and other reasons, to make the composition explicitly at development time.

If the descriptions to be composed are homogeneous - written in the same language - we may be able to mark the correspondences and make the composition without great difficulty. We can compose finite-state machines according to various rules such as sum or difference, marking the correspondences as identities of input letters. We can compose formulae of predicate logic by the logical operators. We can compose grammars in JSP[4,5], marking the correspondences as identities of non-terminals.

But heterogeneous descriptions - written in different languages - are much harder to compose. We may need to compose a recursive function definition with a description of the required interleaving of acceptance of input elements and production of output elements: but the recursive function language gives us no way of marking the correspondences we need. Instead of regarding this, as I believe we should, as a problem demanding improved development tools, we are traditionally inclined to regard it as demanding a more elaborate language[11]. In the same vein, Prolog[12] is an elaboration of a pure logic programming language to include input and output commands masquerading as predicates or goals and an ugly but undisguised sequencing control device in the form of the 'cut'. The penalty we pay for this approach is that the resulting programs in the elaborated language become more complex because the concerns are not separated in any intelligible way.

5.5 In What Order?

Both technical and managerial considerations affect the ordering of description in development. Managers, very properly, want to divide the work among the developers in such a way that there will be the minimum need for communication among them and hence the minimum opportunity for mutual interference and misunderstanding. This is why the simplicities of top-down development seemed so appealing: the chief programmer would write the top level, spawning four or five independent tasks that could be started immediately. Each of these would in turn spawn four or five tasks at the next level, and in no time at all the team of five hundred programmers would be happily working away on independent tasks whose completion would signal the successful end of the project.

The waterfall model of development, in a more global fashion, aims similarly to introduce a combination of managerial and technical discipline. First, the requirements, then the specification, then the implementation. Development progresses monotonically from the 'abstract' concerns of the customer to the 'concrete' concerns of the machine execution.

But if the perspective I am suggesting in this paper has any validity, a system of significant size and complexity is sure to involve constraints on the order of development that will not fit into such simple patterns. We will be forced to proceed quite a long way with one Implementation before we can get very far with a View whose Subject or Target it provides. Perhaps an approximate analysis in the kind of terms I have suggested can point the way to a workable project structure.

Acknowledgement

Daniel Jackson very kindly read earlier drafts of this paper and helped me to make it less unintelligible.

References

[1] T Winograd; Beyond Programming Languages; Comm ACM 22,7 July 1979 pp391-401

[2] D T Ross; Structured Analysis (SA): A Language for Communicating Ideas; IEEE SE Trans 3,1 January 1977 pp16-34

[3] M Weiser; Programmers Use Slices When Debugging; Comm ACM 25,7 July 1982 pp446-452

[4] M A Jackson; Principles of Program Design; Academic Press, 1975

[5] J R Cameron; JSP & JSD: The Jackson Approach to Software Development; IEEE Tutorial Text 1983

[6] M E Conway; Design of a Separable-Transition Compiler; Comm ACM 6,7 July 1963 pp396-408

[7] O-J Dahl and K Nygaard; Simula - an Algol-Based Simulation Language; Comm ACM 9,9 September 1966 pp671-678

[8] E W Dijkstra; The Structure of the THE Operating System; Comm ACM 11,5 May 1968 pp341-346

[9] D L Parnas; On the Criteria to be Used in Decomposing Systems Into Modules; Comm ACM 15,12 December 1972 pp1053-1058

[10] C A R Hoare; Communicating Sequential Processes; Prentice-Hall 1985

[11] J Darlington and L White; The Imposition of Temporal Constraints on the Execution of Term-Rewriting Systems; Imperial College London, September 1986

[12] W F Clocksin and C S Mellish; Programming in Prolog; Springer Verlag 1984