SIMPLICITY AND COMPLEXITY IN PROGRAMS AND SYSTEMS

DRAFT OF 28TH JANUARY 2011

Michael Jackson

The Open University
jacksonma@acm.org

ABSTRACT

The subject of this chapter is behavioural complexity in software development. Complexity is difficulty of human comprehension, whether in analysis or synthesis. Complexity can be regarded as the combination of potentially conflicting simplicities. Reliable development therefore depends on an adequate understanding of both decomposition and combination. The discussion proceeds from complexity in small programs to selected aspects of complexity in realistic computer-based systems. In both settings criteria of simplicity and modes of combination are identified. At the end some general propositions about the understanding and treatment of complexity in software development are recapitulated, followed by a brief final remark on the relationship between understanding and formalism.

1 INTRODUCTION

The topic of this chapter is complexity in an informal sense: difficulty of human comprehension. Inevitably this difficulty is partly subjective. Some people have more experience, or more persistence, or simply more intellectual skill—agility, insight, intelligence, acuity—than others. The difficulty of the subject matter to be mastered depends also on the intellectual tools brought to bear on the task.

These intellectual tools include both mental models and overt models. An overt model is revealed in an explicit public representation, textual or graphical. Its purpose is to capture and fix some understanding or notion of its subject matter, making it reliably available to its original creator at a future time and to other people also. A mental model is a private possession held in its owner's mind, sometimes barely recognised by its owner, and revealed only with conscious effort. A disdain for intuition and for informal thought may relegate a mental model—which by its nature is informal—to the role of a poor relation, best kept out of sight. Such disdain is misplaced in software development.

Complexity is hard to discuss. A complexity, once mastered, takes on the appearance of simplicity. In the middle ages, an integer division problem was insuperably complex for most well-educated Europeans, taught to represent numbers by Roman numerals; today we expect children in primary school to master such problems. Taught a better model—the Hindu-Arabic numerals with positional notation and zero—we learn a fast and reliable route through the maze: its familiarity becomes so deeply ingrained in our minds that we forget why it ever seemed hard to find.

To master a fresh complexity we must understand its origin and its anatomy. In software development a central concern is behavioural complexity, manifested at every level from the behaviour of a small program to the behaviour of a critical system. Behavioural complexity is the result of combining simple behaviours, sometimes drawn from such different dimensions as the program invocation discipline imposed by an operating system, the behaviour of an external engineered electromechanical device, and the navigational constraints of a database.

To master behavioural complexity we must identify and separate its simple constituents, following the second of Descartes's four rules [Descartes 37] for reasoned investigation:

> "... to divide each of the difficulties under examination into as many parts as possible, and as might be necessary for its adequate solution."

But this rule alone is quite inadequate. Leibniz complained [Leibniz]:

"This rule of Descartes is of little use as long as the art of dividing remains unexplained... By dividing his problem into unsuitable parts, the inexperienced problem-solver may increase his difficulty."

So we must devise and apply systematic criteria of simplicity, allowing us to know when we have identified a simple constituent of the complexity that confronts us. But it is not enough to identify the constituent simplicities. We must also understand the origins and anatomy of their existing or desired combination. Developers should not hamper their understanding of a problem by assuming a uniform discipline and mechanism of composition, whether derived from a program execution model or from a specification language.

The complexities to be mastered in software development arise both in tasks of analysis and of synthesis. In analysis, the task is to tease apart the constituents of a given complex whole, identifying each distinct constituent and the ways in which they have been reconciled and brought together. Such analysis may be applied to an existing program, to a requirement, or to any given subject matter of concern. In synthesis the task is to construct an artifact to satisfy certain requirements. For a program, the requirements themselves may be simple and immediately comprehensible: synthesis can then proceed directly. For a realistic computer-based system, requirements are almost always complex, given *a priori* or to be discovered in a process that may be partly concurrent with the synthesis itself. In either case, synthesis can proceed only to the extent that the relevant complexities of the requirement have been successfully analysed and understood.

In this chapter we first consider an example of a small integer program, and go on to discuss small programs that process external inputs and outputs. Then we turn to a consideration of complexities in computer-based systems. At the end of the chapter we recapitulate some general propositions about complexities in software development and techniques for mastering them. The approach throughout is selective, making no attempt to discuss complexity in all its software manifestations, but focusing on complexity of behaviour. In programming, it is this complexity that surprises us when a program that we had thought was simple produces an unexpected result. In a realistic computer-based system, behaviour is harder to understand, and its surprises can be far more damaging. In a critical system the surprises can be lethal.

2 A Small Integer Program

The pioneers of electronic computing in the 1940s recognised the difficulty of the programmer's task. Figure 1 shows a flowchart designed by Alan Turing, slightly modified to clarify a minor notational awkwardness. Turing used it as an illustration in a paper [Turing 49] he presented in Cambridge on 24th June 1949.

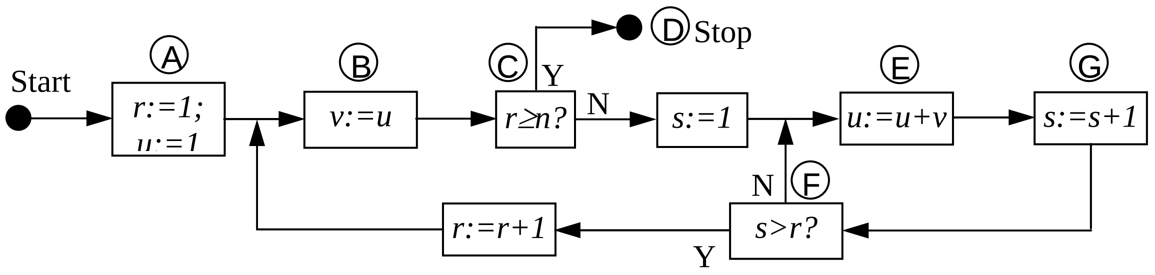

Figure 1. A Flowchart of a Program Designed by Alan Turing

The program was written for a computer without a multiplier. It calculates *factorial(n)* by repeated addition. The value *n* is set in a local variable before the program starts; on termination the variable *v = factorial(n)*. Other local variables are *r*, *s* and *u*. Turing began his talk by asking: How can one check a routine in the sense of making sure that it is right? He recommended that "the programmer should make assertions about the various states that the machine can reach." Assertions are made about the variable values at the entries and exits of the named flow graph nodes. For example, on every entry to node B, $u = r!$; on exit from C to E, $v=r!$, and on exit from C to D, $v = n!$. The program is correct if the assertion on entry to the Stop node is correctly related to the assertion "*n* contains the argument value" on entry to node A from Start.

Along with the flowchart, Turing presented a table containing an entry for each marked block or point in the program: the entry shows “the condition of the machine completely,” including the asserted precondition and postcondition, and the next step, if any, to be executed. The table entries are fragments which can be assembled into a correctness proof of the whole program by checking them in sequence while traversing the flowchart. Further discussion of this program, focusing particularly on the proof, can be found in [Jones 03] and in an interesting short paper [Morris+ 84] by Morris and Jones.

A careful reading of the flowchart shows that the program is essentially structured as an initialisation and two nested loops. The outer loop iterates multiplying by each value from 2 to n; the inner loop iterates to perform each multiplication. However, the flowchart does not express this structure in any systematic way, and Turing’s explanation of the program is difficult to follow. Turing no doubt had a clear *mental* model of the process executed by his program: “multiply together all the integers from *1* to *n* in ascending order”; but his *overt* model of the computation—that is, the flowchart—does not show it clearly. We might even be bold enough to criticise Turing’s program for specific design faults that make it hard to understand. The roles of the variables *u* and *v* are not consistently assigned. On one hand, *v* is the result variable in which the final result will be delivered. On the other hand, *v* is a parameter of the inner loop, specifying the addend by which the multiplication develops its product in the variable *u*. The awkwardness of the exit at block C from the middle of the outer loop is associated with this ambivalence. A further point, made in [Morris+ 84], is that the value of *factorial(0)* is correctly calculated, but this appears almost to be the result of chance rather than design.

Even after a reading of the formal proof has shown the program to be correct in that it delivers the desired result, the program remains complex in the sense that it is hard to understand. One aspect of the difficulty was well expressed by Dijkstra in the famous letter [Dijkstra 68] to the editor of CACM: “we can interpret the value of a variable only with respect to the progress of the process.” Flowcharts offer little or no support for structuring or abstracting the execution flow, and hence little help in understanding and expressing what the values of the program variables are intended to mean and how they evolve in program execution. This lack of support does not make it impossible to represent an understandable execution flow in a flowchart. It means that the discipline inherent in flowcharts helps neither to design a well-structured flow nor to capture the structure clearly once it has been designed.

Such support and help was precisely what structured programming offered, by describing execution by a nested set of sequence, conditional and loop clauses in the form now familiar to all programmers. In the famous letter, Dijkstra argued that this discipline, unlike unconstrained flowcharting, provides useful “coordinates in which to describe the progress of the process,” allowing us to understand the meaning of the program variables and how their successive values mark the process as it evolves in time. Every part, every variable, and every operation of the program is seen in a nested closed context which makes it easily intelligible. Each context has an understandable purpose to which the associated program parts can be seen to contribute; and this purpose itself can be seen to contribute to an understandable purpose visible in the text at the next higher level. These purposes and the steps by which they are achieved are then expressible by assertions that fit naturally into the structure of the text.

This explanation of the benefits of structured programming is compelling, but there is more to say. Structured programming brings an additional benefit that is vital to human understanding. In a structured program text the process, as it evolves in execution, can become directly comprehensible in an immediate way. It becomes captured in the minds of the writer and readers of the text, as a vivid mental model. Attentive contemplation of the text is almost a physical enactment of the process itself; this comprehension is no less vital for being intuitive and resistant to formalisation.

3 PROGRAMS WITH MULTIPLE TRAVERSALS

The problem of computing *factorial(n)* by repeated multiplication is simple in an important respect. The behaviour of Turing’s solution program is a little hard to understand, but this complexity is gratuitous: a more tidily structured version—left as an exercise for the reader—can be transparently simple. Only one simple behaviour need be considered: the behaviour of the program itself in execution. This behaviour can be regarded as a traversal of the factors *1, 2.., n* of *n!,* incorporating each factor into the result by using it as a multiplier when it is encountered in the traversal. The problem world of the program, which is the elementary arithmetic of small integers, imposes no additional constraint on the program behaviour. The argument *n*, the result *v!*, the multipliers, and any local integer values in the other variables can all be freely written and read at will. The program as designed visits the factors of *n!* in ascending numerical order, but descending order is equally possible and other orders could be considered.

More substantial programs, however, usually demand consideration of more than one simple behaviour. For example, a program computing a result derived from an integer matrix may require to traverse the matrix in both row and column order. Both the input and output of a program may be significantly structured, and these structures may restrict the traversal orders available to the program. An input stream may be presented to the program as a text file, or as a time-ordered stream of interrupts or commands. A collection of records in a database, or an assemblage of program objects may afford only certain access paths for reading or writing, and the program must traverse these paths. For example, a program that summarises cellphone usage and produces customer bills must read the input data of call records, perhaps from a database or from a sequential file, and produce the output bills in a format and order convenient for the customers. The traversal of a program's input may involve some kind of navigation or parsing, and production of the output may demand that the records be written in a certain order to build the required data structure.

Multiple behaviours must therefore be considered for input and output traversals. The behaviour of the program in execution must somehow combine the input and output traversals with the operations needed to implement the input-output function—that is, to store and accumulate values from the input records as they are read, and to compute and format the outputs in their required orders. This need to combine multiple behaviours is a primary potential source of software complexity.

A program encompassing more than one behaviour is not necessarily complex if it is well designed. In the cellphone usage example, each customer's call records may be accessible in date order, each giving details of one call; the corresponding output bill may simply list these calls in date order, perhaps adding the calculated cost of each call, and appending summary information about total cost and any applicable discount. It will then be easy to design the program so that it traverses the input, calculates output values, and produces the output while doing so. The two behaviours based on the sequential structures of the two data streams fit together perfectly, and can then be easily merged [Jackson 76] to give the dynamic structure of the program. The program text shows clearly the two synchronised traversals, with the operations on the program's local variables fitting in at the obviously applicable points. The program has exactly the clarity, simplicity, and immediate comprehensibility that are the promised benefits of structured programming.

4 PROGRAMS WITH MULTIPLE STRUCTURES

Sometimes, however, there is a conflict—in the terminology of [Jackson 76], a *structure clash*—between two sequential behaviours both of which are essential to the program. One particular kind of conflict is a *boundary clash*. For example, in a business reporting program, input data may be grouped by weeks while output data is grouped by months. The behaviours required to handle input and output are then in conflict, because there is a conflict between weeks and months: it is impossible to merge a traversal by weeks with a traversal by months to give a single program structure. In a similar example of a different flavour, variable-length records must be constructed and written to fixed length disk sectors, records being split if necessary across two or more sectors. The record building behaviour conflicts with the sector handling behaviour, because the record structure is in conflict with the sector structure. The general form of the difficulty posed by such a conflict is clear: no single structured program text can represent both of the required behaviours in the most immediate, intuitive, and comprehensible way.

To deal effectively with a complexity it must be divided into its simple constituents. In these small programming examples the criterion of simplicity of a proposed division is clear: each constituent behaviour should be clearly described by a comprehensible structured program text. Now, inevitably, a further concern demands attention: How are the simple constituents to communicate? This concern has two aspects—one in the requirement world, the other in the implementation world. One is more abstract, the other more concrete. We might say that one is the communication between behaviours, while the other is the combination of program executions. Here we will consider the communication between the conflicting behaviours. The combination of program executions will be the topic of the next section.

For the business reporting problem, the conflicting behaviours must communicate in terms of days, because a day is the highest common factor of a week and a month: each consists of an integral number of days. Similarly, in the disk sector problem, communication must be in terms of the largest data elements—perhaps bytes—that are never split either between records or between sectors. Ignoring much detail, each problem then has two simple constituent conflicting but communicating behaviours:

- For the business problem: (a) *by-week* behaviour: analysing the input by weeks and splitting the result into days; (b) *by-month* behaviour: building up the output by months from the information by days.
- For the disk sector problem: (a) *by-record* behaviour: creating the records and splitting them into bytes; (b) *by-sector* behaviour: build up the sectors from bytes.

The communication concern in the requirement world demands further consideration, because the constituent behaviours are not perfectly separable. For example, in the processing of monthly business data it may be necessary to distinguish working days from weekend days. The distinction is defined in terms of weeks, but the theme of the separation is to keep the weeks and the months apart. The concern can be addressed by associating a *working/weekend* tag with each day's data. The tag is set in the context of the *by-week* behaviour, and communicated to the *by-month* behaviour. Effectively, the tag carries forward with the day's data an indication of its context within the week. In the same way, the *record* behaviour can associate a tag with each byte to indicate, for example, whether it is the first or last, or an intermediate byte of a record. We will not pursue this detail here.

5 Combining Programs

The program combination concern arises because a problem that required a solution in the form of one executable programmed behaviour has been divided into two behaviours. Execution of the two corresponding programs must be somehow combined in the implementation to give the single program execution that was originally demanded. Possible mechanisms of combination may be found in the program execution environment—that is, in programming language features and in the operating system—or in textual manipulation of the program texts themselves.

The *by-week* and *by-month* behaviours for the business reporting problem communicate by respectively writing and reading a sequential stream of tagged days. An obvious combination mechanism introduces an intermediate physical file of day records on disk or tape. The *by-week* program is run to termination, writing this intermediate file; then the *by-month* program is run to termination, reading the file. This implementation is primitive and simple, and available in every execution environment. But it is also unattractively inefficient and cumbersome: execution time is doubled; use of backing store resources is increased by one half; and the first output record is not available until after the last input record has been read.

In a better combination design, the two programs are executed in parallel, each day record being passed between them to be consumed as soon as it is produced. Having produced each day record, the *by-week* program suspends execution until the *by-month* program has consumed it; having consumed each day record, the *by-month* program suspends execution until the *by-week* program has produced the next day. The two programs operate as *coroutines*, a programming construct first described by Conway as a machine-language mechanism [Conway 63], and adopted as a programming language feature [Dahl+ 72] in Simula 67. In Simula, a program *P* suspends its own execution by executing a *resume(Q)* statement, *Q* being the name of the program whose execution is to be resumed. Execution of *P* continues at the point in its text following the *resume* statement when next another program executes a *resume(P)* statement.

A restricted run-time form of the coroutine combination is provided by the Unix operating system. For a linear structure of programs Unix allows the *stdout* output stream of a program to be either sent to a physical file or piped to another program; similarly, the *stdin* input stream of a program can either be read from a physical file or piped from another program's *stdout*. If the intermediate file of day records is written to *stdout* in the *by-week* program, and read from *stdin* by the *by-month* program, then the Unix shell command

```
InW < ByWeek | ByMonth > OutM
```

specifies interleaved parallel execution of the programs *by-week* and *by-month*, the day records being passed between them in coroutine style.

6 TRANSFORMING A PROGRAM

Conway explains the coroutine mechanism [Conway 63] in terms of input and output operations:

"... each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. ... There is no bound placed by this definition on the number of inputs and outputs a coroutine may have."

From this point of view, the *by-week* program can regard the *by-month* program as an output subroutine, and the *by-month* program can regard the *by-week* program as an input subroutine. If the programming language provides no *resume* statement and the operating system provides no pipes, the developer will surely adopt this point of view at least to the extent of writing one of the two programs as a subroutine of the other. Another possibility is to write both programs as subroutines, calling them from a simple controlling program. These possibilities are pictured in Figure 2.

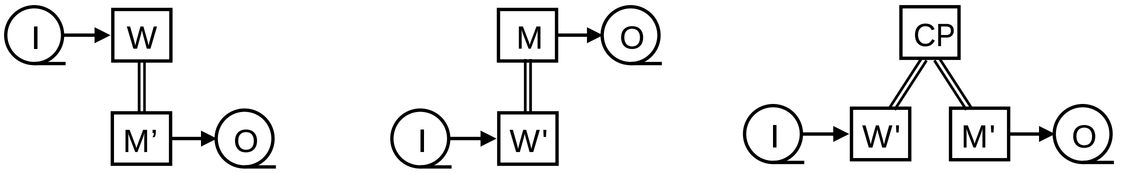

Figure 2. Three ways of Combining Two Small Programs Into One

In the diagrams a tape symbol represents a physical file: I is the input data file; O is the output report file. W and M are the *by-week* and *by-month* programs written as autonomous (or 'main') programs; W' and M' are the same programs written as subroutines; CP is the controlling program, which loops, alternately reading a day record from W' and writing it to M'. A double line represents a subroutine call, the upper program calling the lower program as a subroutine.

The behaviours evoked by one complete execution of the main program W and by one complete sequence of calls to the subroutine W' are identical. This identity is clearly shown by the execution mechanisms of Simula and the Unix pipes, which demand no change to the texts of the executed programs. Even in the absence of such execution mechanisms, the subroutine W' is mechanically obtainable from the program W by a transformation such as *program inversion* [Jackson 76], in which a main program is 'inverted' with respect to one of its input or output files: that it, it is transformed to become an output or input subroutine for that file. Ignoring some details, the elements of the transformation are these:

- a set of labels identifying those points in the program text at which program execution can begin or resume: one at the start, and one at each operation on the file in question;
- a local variable *current-resume-point*, whose value is initialised to the label at the start of the program text, and a switch at the subroutine entry of the form "go to *current-resume-point*";
- implementation of each operation on the file in question by the code:

```
current-resume-point:=X; return; label X:
```

- the subroutine's local variables, including the stack and the *current-resume-point*, persist during the whole of the programmed behaviour.

The essential benefit of such a transformation is that the changes to the text are purely local. The structured text of the original program is retained intact, and remains fully comprehensible. Essentially this transformation was used by Conway in his implementation of coroutines [Conway 63]. Applying the transformation to the development of interrupt-handling routines for a computer manufacturer [Palmer 79] produced a large reduction in errors of design and coding.

Unfortunately, in common programming practice, instead of recognising that W' and W are behaviourally identical, the programmer is likely to see them as different. Whereas the behaviour span of W is correctly seen as the complete synchronised behaviour in which the whole day record file is produced in parallel with the traversal of the whole input data file, the behaviour span of W' is seen as bounded by the production of a single day record. Treating the behaviour span of W' in this way, as bounded by the production of a single day record, casts the behaviour in the form of a large case statement, each limb of the case statement corresponding to some subset of the many different conditions in which a day record could be produced. This is the perspective commonly known as *event-driven programming*. Gratuitously, it is far more complex—that is, both harder to program correctly and harder to comprehend—than the comprehensible structured form that it mistakenly supplants.

7 COMPUTER-BASED SYSTEMS

The discussion in the preceding sections suggests that behavioural complexities in small programs may yield to several intellectual tools. One is a proper use of structured programming in its broadest sense: that is, the capture and understanding of behaviour in its most comprehensible form. Another is the decomposition of a complex behaviour into simple constituent parallel behaviours. Another is the careful consideration of communication between separated behaviours by an identified highest common factor and its capacity to carry any additional detail necessary because the behaviours can be only imperfectly separated. And another is the recognition that the task of combining program executions within an operating system environment is distinct from the task of satisfying the communication requirement between the separated programmed behaviours.

Computer-based systems embody programs, so the intellectual tools for their analysis and development will include those needed for programs. The sources of complexity found in small programs can also be recognised, writ large, in computer-based systems; but for a realistic system there are major additional sources and forms of complexity. These arise in the *problem world* outside the *machine*—that is, outside the computing equipment in which the software is executed. The expression *problem world* is appropriate because the purposes of the system lie in the world outside the machine, but must be somehow achieved by the machine through its interactions with the world. Systems for avionics, banking, power station control, welfare administration, medical radiation therapy and library management are all of this kind. The problem is to capture and understand the *system requirement*, which is a desired behaviour in the problem world, and to devise and implement a behaviour of the computer that will ensure the required behaviour of the world.

The problem world comprises many *domains*: these are the parts of the human and physical world relevant to the system's purposes and to their achievement. It includes parts directly interfaced to the machine through its ports and other communication devices, parts that are the subject of system requirements, and parts that lie on the causal paths between them. Together with the computer, the problem domains constitute a system whose workings are the subject matter of the development. Figure 3 is a sketch of a system to control the lifts in a large building.

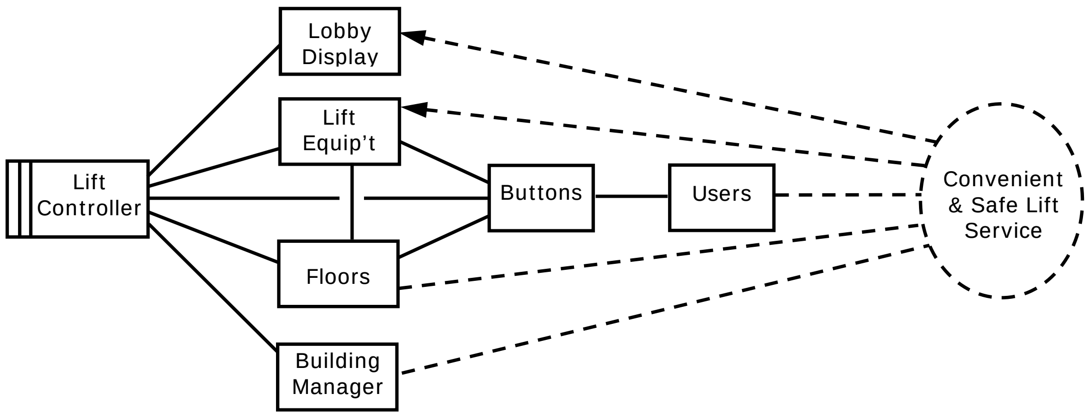

Figure 3: Problem Diagram of a Lift System

The machine is the Lift Controller; plain rectangles represent problem domains; solid lines represent interaction by such shared phenomena as state and events. The dashed oval represents the required behaviour of the whole system. The dashed lines link the oval to the problem domains referenced by the requirement; an arrowhead on a dashed line to a problem domain indicates that the machine must, directly or indirectly, constrain the behaviour of that domain. Here the requirement constrains only the Lobby Display and the Lift Equipment; it refers to, but does not constrain, the Users, the Building Manager (who can specify lift service priorities to suit different circumstances), and the Floors. All problem domains are constrained by their *given properties* and their interactions with other problem domains. For example: by the properties of the Lift Equipment, if the lift direction is set *up*, and the motor is set *on*, the lift car will rise in the shaft; by the properties of the Floors domain the rising car will encounter the floors successively in a fixed vertical sequence. The requirement imposes further constraints that the machine must satisfy. For example: if a user on a floor requests lift service, the lift car must come to that floor, the doors must open and close, and the car must go to the floor desired by the user.

The problem world is an assemblage of interacting heterogeneous problem domains. Their properties and behaviours depend partly on their individual constitutions, but they depend also on the context in which the system is designed to operate. The context sets bounds on the domain properties and behaviours, constraining them further beyond the constraints imposed by physics or biology. For example, the vertical floor sequence would not necessarily be preserved if an earthquake caused the building to collapse; but the system is not designed to operate in such conditions. On the other hand, the system is required to operate safely in the presence of faults in the lift equipment or the floor sensors. If the system is designed for an office building, the time allowed for users to enter and leave the lift will be based on empirical knowledge of office workers' behaviour; in a system designed for an old age home the expected users' behaviour will be different.

8 SOURCES OF COMPLEXITY

The system requirements are complex because they combine several functions. The lift system must provide normal lift service according to the priorities currently chosen by the building manager. Some facility must be provided to allow the building manager to specify priority schemes, to store them, and to select a scheme for current use. The lobby display must be controlled so that it shows the current position and travel direction of each lift in a clear way. A system to administer a lending library must manage the members' status and collect their subscriptions; control the reservation and lending of books; calculate and collect fines for overdue loans and lost books; maintain the library catalogue; manage inter-library loans; and enable library staff to ensure that new and returned books are correctly identified and shelved, and can be easily found when needed.

In a critical system fault-tolerance adds greatly to complexity because it demands operation in different subcontexts within the overall context of the whole system, in which problem domains exhibit subsets of the properties and behaviours that are already constrained by the overall context. The lift system, for example, must ensure safe behaviour in the presence of equipment malfunctions ranging from a stuck floor sensor or a failed request button to a burned-out hoist motor or even a snapped hoist cable. At the same time, lift service—in a degraded form—must be available, subject to the overriding requirement that safety is not compromised.

Further complexity is added by varying modes of system operation. The lift control system must be capable of appropriate operation in ordinary daily use; it must also be capable of operation according to priorities chosen by the building manager to meet unusual needs such as use of the building for a conference. It must also be capable of operating under command of a maintenance engineer, of a test inspector certifying the lift's safety, or of fire brigade personnel fighting a fire in the building.

System functions, or features, are not, in general, disjoint: they can interact both in the software and in the problem domains. In the telecommunications area, *feature interaction* became recognised as a major source of complexity in the early 1990s, giving rise to a series [Reiff-Marganiec and Ryan 2005] of dedicated workshops and conferences. Feature interaction is also a source of complexity and difficulty in computer-based systems more generally. The essence of feature interaction is that features whose individual behaviours are relatively simple in isolation may interfere with each other. Their combination may be complex, allowing neither to fulfil its individual purpose by exhibiting its own simple behaviour. In principle the potential complexity of feature interaction is exponential in the number of features: all features that affect, or are affected by, a common problem domain have the potential to interact.

9 CANDIDATE BEHAVIOUR CONSTITUENTS

In a small program, such as the business reporting program briefly discussed in earlier sections, requirement complexity can be identified by considering the input stream traversal necessary to parse the input data, the output stream traversal necessary to produce the output in the required order, and the input-output mapping that the machine must achieve while traversing the input and output streams. If a structure clash is found, the behaviour is decomposed into simpler constituents, their communication is analysed, and the corresponding programs are combined. Clear and comprehensible simple constituents reward the effort of considering their communication and combination. The approach can be seen as a separation of higher-order concerns: we separate the intrinsic complexity of each constituent from the complexity of composing it with its siblings.

Various proposals have been made for decomposing system behaviour, and have furnished the basis of various development methods:

- *Objects*: each constituent corresponds to an entity in the problem world, capturing its behaviour and evolving state as it responds to messages and receives responses to messages it sends to other objects. For example, in the library system one constituent may capture the behaviour of a library member, another constituent the behaviour of a book, and so on.
- *Machine events*: each constituent corresponds to an event class caused by the machine and affecting the problem world. For example, in the lift system one constituent may correspond to switching on the hoist motor, one to applying the emergency brake, and so on. Each constituent captures an event and the resulting changes in the problem world state.
- *Requirement events*: each constituent corresponds to an event or state value class caused by a problem domain. For example, in the lift system one constituent may correspond to the pressing of a lift button, another to the closing of a floor sensor on arrival of the lift car, and so on. Each constituent captures an event and specifies the required response of the machine.
- *Use cases*: each constituent corresponds to a bounded episode of interaction between a user and the machine. For example, in the library system one constituent may capture the interaction in which a member borrows a book, another the interaction in which a user searches for a book in the library catalogue, and so on. In the lift system one constituent may capture the interaction in which a user successfully summons the lift.
- *Software modules*: each constituent corresponds to an executable textual constituent of the machine's software. For example, in the library system one constituent may capture the program procedure that the machine executes to charge a member's subscription to a credit card, another the procedure of adding a newly acquired book to the library catalogue.

Each of these proposals can offer a particular advantage in some facet or phase of developing a particular system. They are not mutually exclusive, but neither singly nor in any combination are they adequate to master behavioural complexity.

10 Functional Constituent Behaviours

In the famous phrase of Socrates in the Phaedrus, a fully intelligible decomposition of system behaviour must “carve nature at the joints” [Phaedrus 02]. The major joints in a system’s behaviour are the system’s large functions or features. In a decomposition into functions the constituents will be projections of the system and of its overall behaviour.

Each constituent projection of system behaviour has a requirement, a problem world, and a machine; each of these is a projection of the corresponding part of the whole system. To illustrate this idea, Figure 4 shows a possible behaviour constituent of the lift control system.

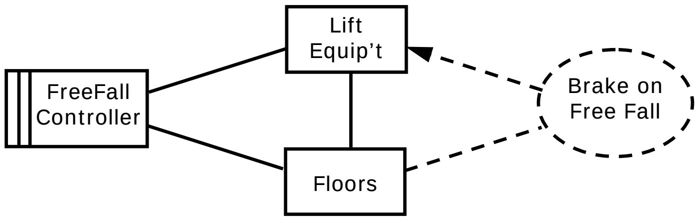

Figure 4: Problem Diagram of a Lift System Constituent

The behaviour constituent shown corresponds to a lift control feature introduced by Elisha Otis in 1852. The lift is equipped with an emergency brake which can immobilise the lift car by clamping it to the vertical steel guides on which it travels. If at any time the hoist cable snaps, the hoist motor is switched off and the emergency brake is applied, preventing the lift car from falling freely and suffering a disastrous impact at the bottom of the shaft. A suitably designed Free Fall Controller might achieve the required behaviour by continually measuring the time from floor to floor in downwards motion of the lift car, applying the brake if this time is small enough to indicate a snapped cable or a major malfunction having a similar effect.

This behavioural constituent is not necessarily a subsystem in the sense that implies implementation by distinct identifiable constituents that will remain recognisable and distinguishable in the complete developed system. In general, the combination of separated simple constituents in a computer-based system is a major task, and must exploit transformations of many kinds. However, for purposes of analysis and understanding, the simple constituents can be regarded as closed systems in their own right, to be understood in isolation from other simple constituents, and having no interaction with anything outside itself. In the analysis, the omitted domains—the Users, Buttons, Lobby Display and Building Manager—play no part. The other behaviours of the Lift Controller machine, too, play no part here: although in the complete system the motion of the lift is under the control of the Lift Controller machine, here we regard the lift car as travelling autonomously in the lift shaft on its own initiative.

By decomposing system behaviour into projections that take the form of subsystems, we bring into focus for each projection the vital question: How can the machine achieve the required behaviour? That is, we are not interested only in the question: What happens? We are interested also in the question: How does it work? To understand each behaviour projection we must also understand its genesis in the workings of the subsystem in which it is defined. This operational perspective affords a basis for assessing the simplicity of each behaviour projection by assessing the simplicity of the subsystem that evokes it. We consider each projection in isolation. We treat it as if it were a complete system, although in fact it is only a projection of the whole system we are developing. This view is far from new. It was advanced by Terry Winograd over thirty years ago [Winograd 79]:

> “In order to successfully view a system as made up of two distinct subsystems, they need not be implemented on physically different machines, or even in different pieces of the code. In general, any one viewpoint of a component includes a specification of a boundary. Behavior across the boundary is seen in the domain of interactions, and behavior within the boundary is

in the domain of implementation. That implementation can in turn be viewed as interaction between subcomponents."

We will turn in a later section to the interactions between distinct constituents. Here we consider the intrinsic complexity—or simplicity—of each one considered in isolation. The criteria of simplicity provide a guide and a check in the decomposition of system behaviour.

11 Simplicity Criteria

Each behaviour constituent, regarded as a subsystem, is what the physical chemist and philosopher Michael Polanyi calls a *contrivance* [Polanyi 58]. A contrivance has a set of characteristic parts, arranged in a configuration within which they act on one another. For us these are the machine and the problem domains. The contrivance has a purpose: that is, the requirement. Most importantly, the contrivance has an *operational principle*, which describes how the parts combine by their interactions to achieve the purpose.

Simplicity of a contrivance can be judged by criteria that are largely—though not, of course—entirely—objective: failure on a simplicity criterion is a forewarning of a development difficulty. The criteria are not mutually independent: a proposed constituent failing on one criterion will probably fail on another also. Important criteria are the following:

- *Completeness*: The subsystem is closed in the sense that it does not interact with anything outside it. In the Free Fall projection the behaviour of the Lift Equipment is regarded as autonomous.
- *Unity of Context*: Different contexts of use demand different modes of operation. An aircraft may be taxiing, taking off, climbing, cruising, and so on. Not all context differences are relevant to all behaviours: differences between climbing and cruising are not relevant to the functioning of the public address system. The context of a simple behaviour projection is constant over the span of the projection.
- *Simplicity of Purpose*: The purpose or requirement of a simple behaviour constituent can be simply expressed as a specific relationship among observable phenomena of its parts. The requirement of the Free Fall constituent is that the emergency brake is applied when the lift car is descending at a speed above a certain limit.
- *Unity of Purpose*: A behaviour projection is not simple if its purpose has the form: "Ensure P1, but if that is not possible ensure P2." This kind of cascading structure may arise in a highly fault-tolerant system. The distinct levels of functional degradation can be behaviour projections.
- *Unity of Part Roles*: In any behaviour constituent each part fulfils a role contributing to achieving the purpose. In a simple behaviour constituent each part's role, like the overall purpose, exhibits a coherence and unity.
- *Unity of Part Properties*: In a simple behavioural constituent each part's relevant properties are coherent and consistent, allowing a clear understanding of how the behaviour is achieved. In a Normal Lift Service behavioural projection, the properties of the Lift Equipment domain are those on which the lift service function relies.
- *Temporal Unity*: A simple behavioural constituent has an unbroken time span. When a behaviour comprises both writing and reading of a large data object, it is appropriate to separate the writing and reading unless they are closely linked in time, as they are in a conversation. In the lift system, the Building Manager's creating and editing of a scheme of priorities should be separated from its use in the provision of lift service.
- *Simplicity of Operational Principle*: In explaining how a behaviour constituent works, it is natural to trace the causal chains in the problem diagram. An explanation of the free fall constituent would trace a path over Figure 4:
    - From the Lift Equipment domain to the Floors domain: "the lift car moves between floors;"
    - At the Floors domain: "lift car arrival and departure at a floor changes the floor sensor state;"

    - From the Floors domain to the Free Fall Controller machine: “the lift car movement is detected by the machine’s monitoring the floor sensors;”
    - At the Free Fall Controller machine: “the machine evaluates the speed of downward movement; excessive speed is considered to indicate free fall”
    - From the Free Fall Controller machine to the Lift Equipment: “if the downward movement indicates free fall the machine applies the brake”.

  Satisfaction of the requirement is explained in a single pass over the causal links, with no backtracking and no fork or join. The complexity of an operational principle is reflected in the number and complexity of the causal paths in the problem diagram that trace out its explanation.

- *Machine Regularity*: The machine in a simple behavioural constituent achieves its purpose by executing a regular process that can be adequately understood in the same way as a structured program.

These criteria of simplicity aim to characterise extreme simplicity, and a developer’s reaction to the evaluation of simplicity must depend on many factors. It remains true in general that major deviations from extreme simplicity warn of difficulties to come.

12 Secondary Decompositions

The simplicity criteria motivate behavioural decompositions beyond those enjoined by recognising distinct system functions. One important general class is the introduction of an *analogic model,* with an associated separation of the writer and reader of the model.

Correct behaviour of a computer-based system relies heavily on monitoring the problem world to detect significant states and conditions to which the machine must respond. In the simplest and easiest cases the machine achieves this monitoring by recognising problem world signals or states whose meaning is direct and unambiguous. For example, in the Lift System the Lift Controller can detect directly that the lift car has arrived at a desired floor by observing that the floor sensor state has changed to *on*.

Often, however, the monitoring of the problem world, and the evaluation of the signals and states it provides, is more complex and difficult, and constitutes a problem that merits separate investigation in its own right. For example, in an employee database in a payroll system, information about the hiring, work and pay of each employee becomes available to the computer as each event occurs. The information is stored, structured and summarised in the database, where it constitutes an *analogic model* of the employee’s attributes, history, and current state. This model is then available when needed for use in calculating pay, holiday entitlement, and pension rights, and also for its contribution to predictive and retrospective analyses. The model, of course, is not static: it is continually updated during the working life of the employee, and its changes reflect the employee’s process evolving in time.

For a very different example, consider a system [Swartout 82] that manages the routing of packages through a tree structure of conveyors. The destination of each package is specified on a bar-coded label that is read once on entry at the root of the tree. The packages are spatially separated on the conveyors, and are detected by sensors when they arrive at each branch point and when they leave. For each package, the machine must set the switch mechanism at each branch point so that the package follows the correct route to its specified destination.

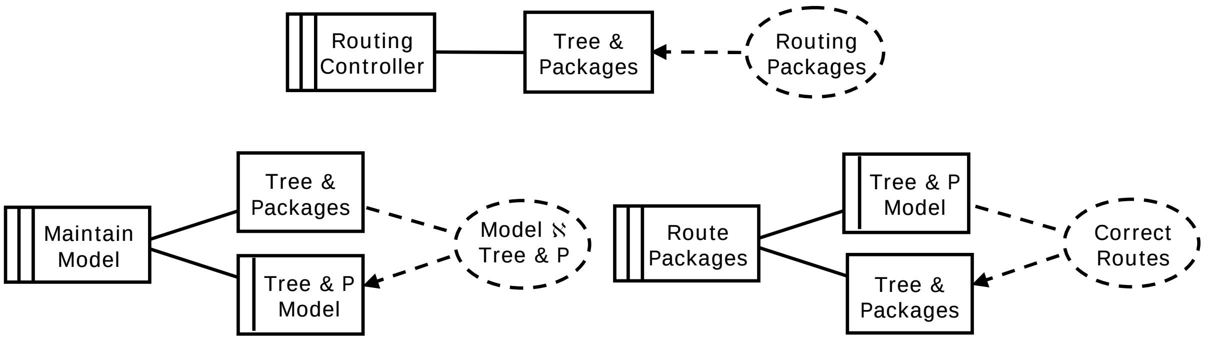

Figure 5: Behaviour Decomposition: Introducing an Analogic Model

The analogic model is needed because although the sensors at the switches indicate that some package has arrived or left, they cannot indicate the package destination, which can be read only on entry to the tree. In the model the conveyors are represented as queues of packages, each package being associated with its bar-coded destination. The package arriving at a switch is the package at the head of the queue in the incoming conveyor; on leaving by the route chosen by the machine, it becomes the tail of the queue in the outgoing conveyor.

The upper part of Figure 5 shows the problem diagram of the whole system; the lower left diagram shows the projection of the system in which the analogic model is built and maintained; the lower right diagram shows the packages being routed through the tree using the analogic model. The analogic model is to be understood as a latent local variable of the Routing Controller machine, exposed and made explicit by the decomposition of the machine's behaviour.

13 The Oversimplification Strategy

A source of system complexity is feature interaction. The complexity of an identified behavioural constituent has two sources. One is the inherent complexity of the constituent considered in isolation; the other is the additional complexity due to its interaction with other constituents. It is useful to separate these two sources. For this purpose a strategy of *oversimplification* should be adopted in initially considering each projection: the projection is oversimplified to satisfy the simplicity criteria of the preceding section. The point can be illustrated by two behaviour constituents in a system to manage a lending library. The library allows its paying members in good standing to borrow books, and the system must manage both membership and book borrowing.

For each member, membership is a behaviour evolving in time. Between the member's initial joining and final resignation there are annual renewals of membership. There are also vicissitudes of payment and of member identity and accessibility: credit card charges may be refused or disputed; bankruptcy, change of name, change of address, promotion from junior to senior member at adulthood, emigration, death, and many other possibly significant events must be considered for their effect on the member's standing.

For each book, too, there is a behaviour evolving in time. The book is acquired and catalogued, shelved, sent for repair when necessary, and eventually disposed of. It can be reserved, borrowed for two weeks, and returned, and a current loan may be renewed before its expiry date. The book may be sent to another library in an inter-library loan scheme; equally, a book belonging to another library may be the subject of a loan to a member. At any point in a book's history it may be lost, and may eventually be found and returned to the library.

A projection that handles both membership and book borrowing cannot satisfy the machine regularity criterion: there is a structure clash between the book and member behaviours. From reservation to final return or loss a loan can stretch over a long time, and in this time the member's status can undergo more than one change, including membership expiry and renewal. So it is desirable to separate the two behaviours, considering each in isolation as if the other did not exist. To isolate the book behaviour we may assume that membership status is constant for each member and therefore cannot change during the course of the member's interaction with the book. The membership behaviour is isolated by assuming that interaction with a book process consists only of the first event of the interaction—perhaps *reserve* or *borrow*. Each process can then be studied and understood in isolation, taking account only of its own intrinsic complexities.

When each behaviour is adequately understood, and this understanding has been captured and documented, their interaction can be studied as a distinct aspect of the whole problem. The questions to be studied will be those that arise from undoing the oversimplifications made in isolating the processes. For example: Can a book be borrowed by a member whose membership will expire during the expected currency of the loan? Can it be renewed in this situation? How do changes in a member's status affect the member's rights in a current loan? How and to what extent is a resigning member to be relieved of membership obligations if there is still an unreturned loan outstanding on resignation? What happens to a reservation made by a member whose status is diminished? The result of studying the interaction will, in general, be changes to one or both of the behaviours.

14 LOOSE DECOMPOSITION

The strategy of oversimplification fits into an approach to system behaviour analysis that we may call *loose decomposition*. Three classes of decomposition technique are pictured in Figure 6. Each picture shows, in abstract form, the decomposition of a whole, A, into parts B, C and D.

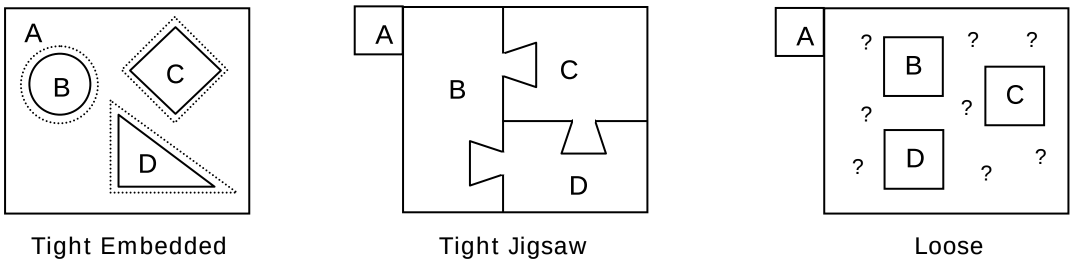

Figure 6: Decomposition Techniques

Embedded decomposition is familiar from programs structured as procedure hierarchies. A is the procedure implementing the complete program; B, C and D are procedures called by A. Each called procedure must fit perfectly, both syntactically and semantically, into its corresponding 'hole' in the text and execution of the calling procedure A. The conception and design of each of the parts B, C and D must therefore simultaneously address any complexity of the part's own function and any complexity arising from its interaction with the calling procedure A and its indirect cooperation, through A, with A's other parts.

Jigsaw decomposition is found, for example, in relational database design. A is the whole database, and B, C and D are tables within it. Essentially, A has no existence except as the assemblage formed by its parts, B, C and D. The parts fit together like the pieces of a jigsaw puzzle, the tabs being formed by foreign keys—that is, by common values that allow rows of different tables to be associated. The process decomposition of CSP is also jigsaw decomposition, the constituent processes being associated by events in the intersection of their alphabets. In jigsaw decomposition, as in embedded decomposition, both the part's own function and its interaction with other parts must be considered simultaneously.

In *loose decomposition*, by contrast, the decomposition merely identifies parts that are expected to contribute to the whole without considering how they will make that contribution or how they will fit together with each other. The identified parts can then be studied in isolation before their interactions are studied and their recombination designed.

In general, it can be expected, as the picture suggests, that there will be gaps to be filled in assembling the whole from the identified parts. Further, the decomposition does not assume that the identified parts can be designed in full detail and subsequently fitted, unchanged, into the whole. On the contrary: the primary motivation for using loose decomposition is the desire to separate the intrinsic complexities of each part's own function from any additional complexities caused by its interaction with other parts. After the parts have been adequately studied, their interactions will demand not only mechanisms to combine them, but also modifications to make the combination possible.

15 RECOMBINING BEHAVIOURS

The purpose of loose decomposition is to separate the intrinsic complexity of each behavioural projection from the complexity added by its interactions with other projections. The recombination of the projections must therefore be recognised as a distinct development task: their interactions must be analysed and understood, and a recombination designed that will support any necessary cooperation and resolve any conflicts. In a spatial dimension, two behavioural projections can interact if their problem worlds include a common domain. In a temporal dimension, they can interact if their behaviour spans overlap or are contiguous.

A very well known recombination problem concerns the potential interference between two subproblem contrivances that interact at a shared problem domain. To manage this potential interference some kind of mutual exclusion must be specified at an appropriate granularity. For interference in a lexical domain such as a database, mutual exclusion is effectively achieved by a transaction structure.

An important class of recombination concern arises when the control of a problem domain is transferred from one subsystem to another. Consider, for example, an automotive system in which the required behaviour of the car while driven on the road is substantially different from its required behaviour when undergoing a regular servicing. If the two behaviours have been separated out into two behaviour projections, then at some point when the car is taken in for servicing, or, conversely, taken back from servicing to be driven on the road, control of the car must pass from one to the other. The former, currently active, subproblem machine must suspend or terminate its operation, and the latter, newly active, must resume or start. The problem of managing this transfer of control has been called a *switching concern* [Jackson 01].

The focus of a switching concern is the resulting concatenated behaviour of the problem world. This concatenated behaviour must satisfy two conditions. First, any assumptions about the initial problem world state on which the design of the latter contrivance depends must be satisfied at the point of transfer. For example, in the automotive system the latter subproblem design might assume that the car is stationary with the handbrake on, the engine stopped, and the gear in neutral. Second, the concatenated behaviour must satisfy any requirements and assumptions whose scope embraces both the former and the latter subproblem.

Two behaviour projections' lifetimes may be coterminous: for example, the free fall constituent is always in operation and so is the constituent that displays the current location of the lift car. In general, the operational lifetimes of distinct subproblem contrivances are not coterminous. One may begin operation only when a particular condition has been detected by another that is monitoring that condition: for example, a contrivance that shuts down the radiation beam in a radiotherapy system may be activated only when the emergency button is pressed. A set of subproblem contrivances may correspond to successive phases in a defined sequential process: for example, taxi, take-off, climb, and cruise in an avionics system. One contrivance's operational lifetime may be nested inside another's: for example, a contrivance that delivers cash from an ATM and the contrivance that controls a single session of use of the ATM.

In discussing small programs we distinguished the required communication between separated simple constituents from recombining their execution to fit efficiently into the operational environment. For computer-based systems, the recombining the execution of separated simple behaviours is a large task in its own right, often characterised as software architecture.

16 SOME PROPOSITIONS ABOUT SOFTWARE COMPLEXITY

This section recapitulates some propositions about software complexity, summarising points already made more discursively in earlier sections.

(a) Success in software development depends on human understanding. We perceive complexity wherever we recognise that we do not understand. Complexity is the mother of error.

(b) Behavioural complexity is of primary importance. A complex behaviour is a combination of conflicting simple behaviours. In analysis we identify and separate the constituent simple behaviours. In synthesis we clarify their communication and recombine the execution of the programs that realise them.

(c) For small programs there are three obvious categories of required behaviour: traversing the inputs—that is, parsing or navigating them; traversing the outputs—that is, producing them in the required order and structure; and computing the output data values from the input.

(d) In each category of required behaviour of a small program, a behaviour is simple if it can be represented by a labelled regular expression, as it is in a structured program text. In general, a structured program is more understandable than a flowchart.

(e) A structured program is understandable because it localises the demand for understanding at each level of the nested structure. More importantly, the described behaviour is comprehensible in an intuitive way that is closely related to a mental enactment of the behaviour. The importance of this comprehension is not lessened by its intuitive nature, which resists formalisation.

(f) Complexity in a small program can be mastered by separating the conflicting behaviours into distinct simple programs. Communication between these programs demands explicit clarification and design because they may be only imperfectly separable. This design task is concerned to satisfy the behaviour requirement.

(g) The task of combining simple program executions is concerned with implementation within the facilities and constraints of the programming language and execution environment. Parallel execution facilities such as coroutines or Unix pipes may make this task easy.

(h) In the absence of parallel execution facilities the simple programs must often be combined by textual manipulation. Systematic manipulation can convert a program into a subroutine with persistent state; this subroutine can then play the role of an input routine for one of its output files, or an output routine for one of its input files.

(i) Requirements for a computer-based system stipulate behaviours of the problem world. The system is an assemblage of interacting heterogeneous parts, or domains, including the machine, which is the computer equipment executing the software.

(j) The software development problem for a system includes: clarifying and capturing the requirements; investigating and capturing the given properties and behaviours of the problem domains; and devising a behaviour of the machine to evoke the required behaviour in the problem world.

(k) Realistic systems have multiple functions, operating in various modes and contexts. These functions, modes and contexts provide a basic structure for understanding the system behaviour.

(l) Like a complex behaviour of a small program, a complex behaviour of a system is a combination of simple behaviours, each a projection of the whole. Each is a behaviour of an assemblage of problem domains and the machine. These simple behaviours can interact both within the machine and within common problem domains.

(m) For a system, the behaviours of interest are not input or output streams or computing the values of output from inputs. They are joint behaviours of parts of the problem world evoked by the machine. They must therefore be understood as behaviours of contrivances, comparable to the behaviours of such mechanical devices as clocks and motor cars.

(n) In addition to its interacting parts, a contrivance has a purpose and an operational principle. The purpose is the behavioural requirement to be satisfied by the contrivance. The operational principle explains how the purpose is achieved: that is, how the contrivance works. Understanding of the operational principle is essentially an informal and intuitive comprehension, resistant to formalisation.

(o) Some criteria of simplicity in a contrivance can be understood as unities: unity of requirement; unity of the role played by each domain in satisfying the requirement; unity of context in which the contrivance is designed to operate; unity of domain properties on which the contrivance depends; and unity of the contrivance's execution time.

(p) An overarching criterion is simplicity of the operational principle. Any operational principle can be explained by tracing the operation along causal links in the configuration of domains and their interactions. An operational principle is simple if it can be explained in a single pass over the configuration, with no backtracking and no fork or join.

(q) As in a small program, a criterion of simplicity for a contrivance is that the behaviour of the machine can be adequately represented by a labelled regular expression, as it is in a structured program text.

(r) The criteria of simplicity enjoin further decompositions. In particular, many system functions can be decomposed into the maintenance of a dynamic model of some part of the problem world, and the use of that model. Similarly, where the system transports data over time or place or both, the writing should be separated from the reading.

(s) Communication between separated behaviours, and combination of the executions of the machines that evoke them, are a major source of complexity in systems. Loose decomposition is therefore an effective approach: consideration of communication and combination is deferred until the constituent behaviours are well enough understood.

(t) Because separation into simple behaviours can rarely be perfect, understanding of constituent behaviours usually demands initial oversimplification. The oversimplification can be reversed later, when the communication between the simple behaviours is considered.

(u) For a system, combining the machine executions of constituent behaviours is—or should be—the goal of software architecture after the constituent behaviours have been adequately understood.

17 Understanding and Formalism

The discussion of software complexity in this chapter has focused on human understanding and has ignored formal aspects of software development. Formal reasoning, calculation, and proof are powerful tools, but they are best deployed in the context of an intuitive, informal, comprehension that provides the necessary structure and guiding purposes. Polanyi stresses the distinction between science and engineering [Polanyi 66]:

> "Engineering and physics are two different sciences. Engineering includes the operational principles of machines and some knowledge of physics bearing on those principles. Physics and chemistry, on the other hand, include no knowledge of the operational principles of machines. Hence a complete physical and chemical topography of an object would not tell us whether it is a machine, and if so, how it works, and for what purpose."

A similar distinction applies to software development and formal mathematical reasoning. The historic development of structured programming illustrates the point clearly. Rightly, the original explicit motivation was human understanding of program executions. Later it proved possible to build on the basis of the intuitively comprehensible program structure. Correctness proofs exploited this structure, using loop invariants and other formal techniques. This is the proper role of formalism: to add strength, precision and confidence to an intuitive understanding. Unfortunately, advocates of formal and informal techniques often see each other as rivals. It would be better to seek means and opportunities of informed cooperation in the mastery of software complexity.

References

[Conway 63] Melvin E Conway; *Design of a Separable Transition-Diagram Compiler*; Communications of the ACM Volume 6 Number 7, pages 396-408, July 1963.

[Dahl+ 72] Ole-Johan Dahl and C A R Hoare; *Hierarchical Program Structures*; in Ole-Johan Dahl, E W Dijkstra and C A R Hoare; Structured Programming; Academic Press, 1972.

[Descartes 37] Rene Descartes; *Discourse on the method of rightly conducting the reason, and seeking truth in the sciences*; 1637.

[Dijkstra 68] E W Dijkstra; *A Case against the GO TO Statement*; EWD215, published as a letter (*Go To Statement Considered Harmful*) to the editor of Communications of the ACM Volume 11 Number 3, pages 147-148, March 1968.

[Jackson 76] M A Jackson; *Constructive Methods of Program Design*; in Proceedings of the 1st Conference of the European Cooperation in Informatics, pages 236-262; G Goos & J Hartmanis eds; Springer-Verlag LNCS 44, 1976.

[Jackson 01] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2001.

[Jones 03] Cliff B. Jones; *The Early Search for Tractable Ways of Reasoning about Programs*; IEEE Annals of the History of Computing Volume 25 Number 2, pages 26-49, April 2003.

[Leibniz 90] G W Leibniz; *Philosophical Writings (Die philosophischen Schriften)* Volume IV, C I Gerhardt ed, page 331, 1857-1890.

[Morris+ 84] F L Morris and C B Jones; *An Early Proof by Alan Turing*; Annals of the History of Computing Volume 6 Number 2, pp139-143, April 1984.

[Palmer 79] P F Palmer; *Structured programming techniques in interrupt-driven routines*; in ICL Technical Journal Volume 1 Number 3, pages 247-264, November 1979.

[Phaedrus 02] Plato; *Phaedrus*; translated by Robin Waterfield; Oxford University Press World's Classics, 2002.

[Polanyi 58] Michael Polanyi; *Personal Knowledge: Towards a Post-Critical Philosophy*; Routledge and Kegan Paul, London, 1958, and University of Chicago Press, 1974.

[Polanyi 66] Michael Polanyi; *The Tacit Dimension*; pages 39-40, University of Chicago Press, 1966; republished with foreword by Amartya Sen, 2009.

[Reiff-Marganiec and Ryan 05] Stephan Reiff-Marganiec and Mark D. Ryan eds; *Feature interactions in telecommunications and software systems VIII*; IOS Press, 2005.

[Swartout 82] William Swartout and Robert Balzer; *On the Inevitable Intertwining of Specification and Implementation*; Communications of the ACM Volume 25 Number 7, pages 438-440, July 1982.

[Turing 49] A M Turing. *Checking a large routine*; In Report on a Conference on High Speed Automatic Calculating Machines, pages 67-69, Cambridge University Mathematical Laboratory, Cambridge, 1949. Also discussed in [Morris+ 84, Jones 03].

[Winograd 79] Terry Winograd; *Beyond Programming Languages*; Communications of the ACM Volume 22 Number 7, pages 391-401, July 1979.

Chapter 2 in Mike Hinchey and Lorcan Coyle eds, Conquering Complexity, Springer Verlag, 2012