The Operational Principle and Problem Frames

(Draft of 13 August 2009)

Michael Jackson, The Open University
jacksonma@acm.org

Abstract In the problem frames approach to software development—as its name indicates—analysis of the *problem* precedes construction of the *solution*. The problem analysis rests on certain ideas of structure and simplicity, including a general recommendation that composition should be postponed until the parts to be composed are well understood in their preliminary isolated forms. These ideas are discussed in the light of Michael Polanyi's notion of the *operational principle* of a machine or *contrivance*, and his account of the relationship between scientific knowledge and understanding of machines. Criteria are suggested for simplicity in problem decomposition. The outline structure of the associated development approach is sketched, and the relationship between formal development methods and problem structuring is clarified.

1. Complexity and Simplicity

In software development complexity is the mother of failure. As C A R Hoare famously said, speaking of Algol W and Algol 68 in his Turing Award lecture [Hoare81]: "I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies." The emphasis on what is *obvious* shows clearly that Hoare was talking about complexity in a subjective human sense—about a particular kind of obstacle to human understanding. It is intellectual complexity, that manifests itself as difficulty in designing, writing and understanding software—in developing a dependable system and achieving confidence that it will provide the required functionality.

1.1 Computer-Based Systems

The theme of this paper is mastery of this kind of complexity as it is encountered in software development for *software-intensive* or *computer-based systems*. These are systems in which computers interact with the physical world in order to bring about certain desired effects there. For example: a system for a lending library, intended to control library membership and the acquisition, cataloguing and lending of books, and to provide information about these activities; an avionics system, designed to help the pilot to fly the plane safely and efficiently; or a lift control system, whose purpose is to provide convenient and safe transport from floor to floor in a tall building. In all of these systems, the success of the development is judged by the effects in the physical world. The true subject matter of the software development activity is not the computations carried out inside the computer, but the desired behaviour that these computations evoke and control in the world outside.

Realistic systems of this kind are complex. The world with which the computer interacts is usually a heterogeneous assemblage of physical *domains*. For the avionics system these domains include the earth's atmosphere, the pilot, the airport runways, the aircraft's engines and its control surfaces. For the lending library system they include the library staff, the members, the books and the barcode labels stuck on their covers, and the membership swipe cards. For the lift-control system the domains include the lift shafts and the building's floors, the lift cars, the request buttons, the users, the lift and lobby doors, the lift position display, and so on. Further complexity arises from the proliferation of features, along with their inescapable feature interactions, in response to market pressures. At the same time, the highest possible level of automation is sought, even within critical systems. The development process, then, must address the software's interactions with many physical domains of different natures, exploiting the multiple properties of each domain in the service of each of many interacting features. It is in the understanding and development of such complex systems that we seek simplicity.

Much of the complexity in these systems springs from the interactions of relatively simple constituents brought together to form complex wholes. Mastering this complexity demands the unravelling—and subsequently the analysis and reconstruction—of these interactions. Formal languages, analysis, reasoning and calculation are vital tools in this task; and so too is a sound technique of formalising a non-formal reality so that it can be reasoned about as reliably as possible. But these tools alone are not sufficient. They must be applied within a conceptual framework which supports and guides both a decomposition of a complex whole into simple parts and an analysis of the interactions among those parts that must be accommodated when they are recombined into the desired whole. Proposing and clarifying such a framework is the purpose of this paper.

1.2 Problem Frames and the Operational Principle

The conceptual framework proposed in this paper is that of the problem frames approach [Jackson01] to software development. Development of a system is regarded as a *problem*: the task is to devise a software behaviour that will satisfy the *requirement*—that is, will produce the required effects in the physical *problem world*. The complexity of the problem is addressed by decomposition into *subproblems*, and so on recursively, until the subproblems obtained are sufficiently simple to be understood and solved without further decomposition. A subproblem is simple when the argument necessary to justify a solution is itself simple by certain specific criteria.

Each simple subproblem can be regarded as defining a small system to be developed—with its own software behaviour, problem world, and requirement. The subproblem and its associated system are regarded as closed: they are to be analysed in isolation, temporarily ignoring interactions with other subproblems. The subsequent recombination of the analysed subproblems, to give the analysis of the original whole problem, is a substantial task in its own right. Naturally, the decomposition into simple subproblems proceeds top-down, while their recombination proceeds bottom-up.

Support for the ideas on which this conceptual framework is based, and clarification of its consequences, can be drawn from the notion of the *operational principle* of a machine or *contrivance*, extensively discussed by the philosopher and physical chemist Michael Polanyi in his book Personal Knowledge [Polanyi58]. Each small system defined by a simple subproblem can be regarded as a contrivance in Polanyi's sense, the software and the problem domains constituting the characteristic parts of the contrivance. Polanyi lays great stress on the human and individual nature of knowledge and understanding, which are central to the practical work of developing a computer-based system. He emphasises the distinction between scientific knowledge and the understanding of machines. This distinction has a clear parallel in software development: formal scientific and mathematical knowledge are to be distinguished from the understanding of problem and system structures within which they can be deployed. The relationship between them is discussed here in the context of the conceptual framework of the problem frames approach.

1.3 A Caveat

A caveat is necessary before the substance of the paper is presented in the sections that follow. Success in software development, as in any human activity, is often to be sought by identifying and following successful precedents—in short, by practising *normal design* as discussed by the aeronautical engineer Walter Vincenti [Vincenti93]. The focus of the development work is then the instantiation and improvement of an existing accepted design: the developer rarely has reason to reconsider the decomposition into components or to devise a new configuration of the components and their interactions. In effect, the long evolution of the normal design has gradually stimulated, and then absorbed, successive steps in a mastery of the problem complexities: in normal practice it is unnecessary to recapitulate that evolution.

In this paper, however, mastery of complexity is the central topic. The discussion will therefore implicitly assume that the development problem is—at least to a large extent—novel, and that the developer cannot rely chiefly on precedent but must draw on general principles to master the problem complexity. The practical justification for adopting this assumption is that many areas of software development are regrettably lacking in established and acknowledged normal designs. The intellectual justification is that mastery of complexity *ab initio* is a topic of intrinsic interest.

2. THE OPERATIONAL PRINCIPLE

Polanyi elaborates the notion of an operational principle as it applies in several fields. He discusses operational principles in language, mathematics, biology, psychology and even in logic. He also applies and illustrates it in the field of *machines* such as clocks, locomotives, telephones and cameras, and other *contrivances* of a like nature, which are assemblages of physical parts. This is where the notion is most directly applicable to the development of computer-based systems and to the theme of this paper.

2.1 A Machine Example

The operational principle of a machine specifies [Polanyi58] how “its characteristic parts—its organs—fulfil their special function in combining to an overall operation which achieves the purpose of the machine”. For example, we may describe the operational principle of the weight-driven pendulum clock in terms of the weights, the gear train, the hands, the escapement, the pendulum, and the passage of time. In the style of a *problem diagram* [Jackson01], the interactions among these parts, together with the purpose of the whole machine, are sketched in Figure 1:

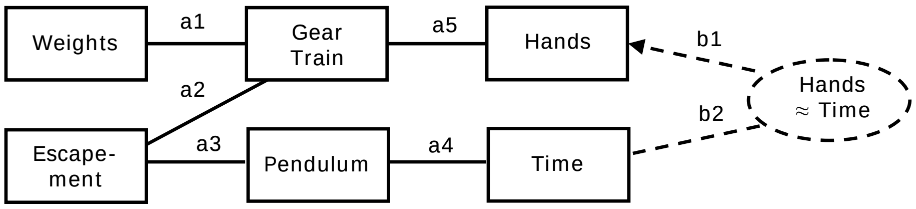

Figure 1. A Pendulum Clock and its Purpose

The operational principle is as follows. The weights, under gravity, apply power (a1) to turn the gear train. The escapement wheel turns (a2), being fixed to a shaft in the gear train. The pendulum is connected to the escapement lever. Through the lever each swing of the pendulum allows the escapement wheel to advance by one unit for each swing (a3) and also to give a small impulse (a3) to keep the pendulum swinging. The gear train shafts therefore turn proportionally to the number of swings of the pendulum, and the rotating hands, fixed to appropriate shafts in the train (a5), count the pendulum swings (a4) and so effectively indicate the passage of time. The purpose of the machine is shown in the dashed oval: it is to govern the angular positions of the hands (b1) to correspond to the elapsed time (b2). The arrowhead on the dashed line b1, and its absence from the line b2, indicate that the purpose of the contrivance is to constrain the hands, not to constrain the passage of time.

2.2 Science and the Operational Principle

Polanyi is at pains to stress the difference between knowledge and understanding of the operational principle of a contrivance, and knowledge of the relevant natural science and mathematics. He goes so far as to write:

> "... Indeed, the understanding of the structure and operation of a machine require as a rule very little knowledge of physics and chemistry. Hence the two kinds of knowledge, the technical and the scientific, largely by-pass each other.
>
> "But the relation of the two kinds of knowledge is not symmetrical. If any object—such as for example a machine—is essentially characterised by a comprehensive feature, then our understanding of this feature will grant us a true knowledge of what the object is. It will reveal a machine as a machine. But the observation of the same object in terms of physics and chemistry will spell complete ignorance of what it is. Indeed, the more detailed knowledge we acquire of such a thing, the more our attention is distracted from seeing what it is."

The "understanding of the structure and operation of the machine" does not itself explain in mathematical and scientific terms the detailed conditions necessary for the clock to achieve its purpose. Rather, it provides an intellectual and practical structure within which such an explanation can be formulated and given. This explanation must rest on two foundations. First, on a scientific understanding of the physical properties of the characteristic parts of the clock. The pendulum swings with an approximately constant period that depends—in accordance with the mathematical analysis of the forces acting on it—on its length and the acceleration due to gravity. The impulses imparted to the pendulum are strong enough to compensate for the effects of friction and air resistance. The weight is heavy enough to drive the whole mechanism. The escapement mechanism exploits the mechanical principle of the lever to minimise disturbance of the pendulum and wear on the contact surfaces. The gear ratios of the shafts for the hands and the escapement are correctly matched to the pendulum's period—and so on. Second, the scientific explanation depends for its applicability on the assumed *context* of the machine operation. The clock must be located in the earth's gravitational field; it must be positioned close to sea level; it must be stably located on *terra firma* and not tossed about on a ship at sea; the centre of the pendulum swing must be at the vertical position; and so on. The scientific and mathematical explanations given need be valid only in the specific local context for which the machine has been devised. If this local context is familiar, or readily understood, the contrivance and its operational principle are easily grasped: they provide the intellectual structure for the scientific and mathematical explanations and for their understanding and validation.

3. The Operational Principle in Computer-Based Systems

Polanyi's discussion returns more than once to the description of the contrivance as it would be—or is—presented in a patent, because he sees the patent claim as a document in which the inventor will "...always try to obtain a patent in the widest possible terms; he will therefore try to cover all conceivable embodiments of its operational principle by avoiding the mention of the physical or chemical particulars of any actually constructed machine, unless these are strictly indispensable to the operations claimed for the machine.” The patent applicant, in short, is trying to give the most abstract specification of the invention that is consistent with the conditions on which patents can be issued, excluding inessential implementation detail that would limit the scope of the patent.

3.1 The Given Problem World

Practical software development, unlike a patent application, is usually concerned with operational principles in a very specific concrete form. A computer-based system brings together the *software*, executed by one or more computers, with a *problem world*—which is a heterogeneous assemblage of physical *problem domains*. The purpose of the software[1] is to govern the interactions of the computers with the world and, through these interactions, to achieve some observable effects in the world: these observable effects are the *functional requirement* for the system.

In the design of a contrivance such as a clock, the designer is in principle free to choose any assemblage of parts, and to arrange any interaction paths among them, that can achieve the purpose of the contrivance. In a computer-based system the parts are the computer (executing the software to be developed) and the problem domains, and the interaction paths are the interfaces of phenomena they share. For software development *per se*, this configuration of parts and interfaces is largely—or even entirely—given: that is, it is not open to the software developers to introduce additional or replacement problem domains, or to introduce new interaction paths between problem domains or between the computer and the problem domains. These given parts and interfaces determine the arrangement of the system’s parts in a very concrete way. The problem to be solved by the software developers is to devise a software behaviour that will achieve the system’s purpose within this given configuration.

3.2 A Zoo Turnstile

Figure 2 depicts a very small example. The system is intended to control entry to a zoo, ensuring that each visitor pays for entry.

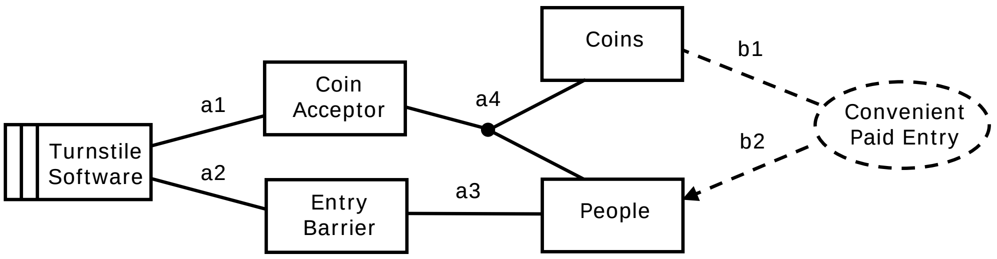

Figure 2. Controlling a Zoo Turnstile

Entry is protected by a turnstile barrier controlled by the software, which is also interfaced to a coin acceptor into which entry fees are to be inserted. The parts of the system are the given problem domains—the People, the Coin Acceptor, the Coins, and the Entry Barrier—and the Turnstile Software, which is to be developed. These parts and their interfaces are identified in Figure 2: people may insert coins into the acceptor (a4); they may enter through the barrier when the barrier allows entry (a3); the software controls the entry barrier (a2) and detects coins inserted into the acceptor (a1). The purpose of the system, represented by the

[1] In the usual presentation of the problem frames approach [Jackson01] the software and computer are regarded as together constituting the *machine*. Here we avoid this term because Polanyi uses it for what we consider to be the whole system, comprising both the software and the problem world.

dashed oval, is to achieve *Convenient Paid Entry*, by controlling people's entry (b2) with respect to the insertion of coins (b1); the insertion of coins (b1) is not constrained. The operational principle is readily expressed in terms of the system parts. People can insert coins (a4) into the acceptor; the turnstile software detects (a1) coin insertions and releases the entry barrier (a2) accordingly to allow people to enter (a3).

Evidently, within this operational principle there is room for variation in the purpose of the system and in the realisation of the operational principle, even when the problem world is given. For example, depending on the relative positions of the coin acceptor and entry barrier, there may be neither the intent nor the possibility of ensuring that each person admitted is the person who most recently inserted a coin. The purpose must then specify which possible interleavings of coin insertion and barrier release events are acceptable. There are arguments in favour of strict alternation; but a schoolteacher equipped with a handful of coins to pay for pupils on a school outing may be grateful for a looser scheduling that ensures only that cumulatively the number of entries does not exceed the number of coins inserted.

3.3 Understanding the Operational Principle

Although there is room for variation, the system's purpose and its operational principle are simple enough to be humanly intelligible. This simplicity is vital, because it provides a clear structure within which the detailed examination and analysis can be made of the given domain properties, and of the possibilities they offer for satisfying the system's functional requirements. Polanyi boldly asserts:

> "Unless I believe a purpose to be reasonable or at least conceivably reasonable, I cannot endorse an operational principle which teaches how to achieve this purpose."

Putting the same point differently: the developers will be more liable to confusion and error in their work unless they have both accepted the functional requirement and clearly understood the operational principle of the system by which the requirement is to be achieved. Although the eventual development will embrace many details that must be dealt with exactly, the acceptance and understanding spoken of here do not depend on exactness. They depend more on the recognition of components of familiar kinds, interacting in familiar structures and configurations. It is with this clear recognition that the detailed work of modelling the problem world domains and their interactions, and of devising a satisfactory software behaviour, can be most reliably conducted.

3.4 Solving the Problem

Solving the software development problem depends on detailed investigation, formalisation and analysis of the properties and behaviours of the problem domains, and of the shared phenomena by which they interact with each other and with the software. For example, it is necessary to understand not only the interface (a2) at which the software can control the entry barrier, but also the interactions at (a3) between a visitor trying to enter and the possible states of the barrier. To enter it may be necessary to *push* on the barrier, indicating at (a2) that entry is requested; if the software then *releases* the barrier (a2) the visitor can *enter* by pushing the barrier further, whereupon the barrier reverts to its *locked* state to prevent a further entry until there has been a further pair of push and release events. This barrier behaviour is operationally similar to the behaviour of the clock escapement, and may similarly involve some matters of timing in the interactions of the system's parts.

When the development is complete, the developers must be able to show that they have produced an adequate solution. This adequacy argument will involve formal descriptions of the requirement and of the properties and behaviours of the problem domains and the software. For example, it may include a finite state machine description of the given behaviour of the entry barrier, from which the effects at (a3) of possible visitor behaviours at (a3) combined with the software's control behaviour at (a2) can be formally deduced. The operational principle of the system gives an outline structure for the adequacy argument. The argument will include reasoning along causal chains in the system, both within the software and the problem domains and at their interaction interfaces. The formal descriptions of properties and behaviours of the software and of problem domains, and the reasoning based on them, furnish the lemmas that allow some detail to be hidden and the structure of the whole argument to be clearly visible. Local invariants may provide succinct links between behaviours of adjacent domains. Some aspects of the system requirements may be captured formally in global invariants that can be shown to hold over the operation of the whole system. In essence, the role of the adequacy argument is to show that the system embodies its operational principle, not only in the large, but in the small also; and that it does so in a way that achieves its purpose.

4. PROBLEMS AND SOLUTIONS

The zoo turnstile system is unrealistically small and simple, permitting a simple relationship between the problem and its solution. The functional requirement—the purpose—of the system can be easily understood and tersely expressed, and its operational principle is easily grasped and easily referred to the behaviour of the problem domains and the software.

4.1 Refinement

Development of the solution can proceed by a kind of refinement, successively stepping across the problem diagram of Figure 2 from right to left, appealing at each step to the given properties of the relevant problem domain [JacksonZave95]. The requirement is expressed in terms of coin insertions (b1) and entries (b2). The coin insertions are elementary events (a4), but each entry event (a3) is the culmination of a little protocol executed by the visitor and the entry barrier. The given properties of the coin acceptor allow the coin insertions to be refined to events at the interface (a1) between the software and the coin acceptor. The given properties of the turnstile allow the entry protocol for the visitor to be refined to a protocol of events at the interface (a2) between the software and the entry barrier. In the final refinement steps the software itself can be developed in detail to maintain the required relationship between events at (a1) and events at (a2).

Subject to developer trial and error, this kind of development is a monotonic progression from problem to solution, refining and elaborating the requirement until it becomes the software solution. The structure of the requirement is elaborated to respect and exploit the given problem world properties (which must, of course, be explicitly described in documentation referenced in the refinement steps), and becomes the structure of the solution. There is no well-defined point in the development at which the developer shifts attention from problem analysis to construction of the solution.

4.2 Limitations of Refinement

The advantages of refinement as a technique for developing a program from a formal program specification are well known. Here we are arguing that a form of refinement can also be used in developing the software of a small and simple system. However, it is not fully and directly applicable to realistically large and complex computer-based systems. The immediately obvious obstacle is that neither the operational principle nor the functional requirement of such a system can be captured in a terse formal specification, even at a high level of abstraction. What is the purpose of the global telephone network system? Of the lending library system? Of an avionics system? Of a chemical process control system? Of a banking system?

Of course, it is easy enough to choose some salient, centrally important, function. “The purpose of the telephone system is to enable people to talk to each other,” we may say. But such a large purpose, unlike the turnstile requirement, gives no useful purchase on the first refinement step. It is simultaneously too large to grasp as a development objective, and too small to encompass the whole of the required functionality. What about billing? What about conference calls, call forwarding and wake-up calls? Or automatic callback, call blocking and credit card calls? Some decomposition and structuring of the purpose or requirement must take place before it can form a basis for solution development.

4.3 Problem Structuring

In the problem frames approach, structuring the purpose or requirement of the system is regarded as structuring the *problem*. The problem is regarded as having the general form exemplified by Figure 2. That is: it defines a small system to be developed, having a problem world comprising given problem domains, a software part, interaction paths among them, and a system functional requirement, which stipulates the effects to be brought about in the problem world by the execution of the software. If the requirement is very complex, as it will be in a realistic computer-based system, then the problem must be decomposed into subproblems, each subproblem itself having the general form of a problem. Decomposition continues recursively until the problems at the leaves of the decomposition tree have clear and easily understandable purposes and operating principles, and are sufficiently simple—like the turnstile problem—to be solved directly.

This is a decomposition of the problem, not of the solution. The small systems defined by the subproblems are not, in general, structural *subsystems* of the whole system to be built. Nor can the problem structure that results from the decomposition be confidently expected to serve as a solution structure: the software in each subproblem is not expected to become a module of the software of the whole system when it is eventually completed. Rather, the small systems defined by the subproblems should be regarded as *projections* of the whole system. The behaviours of the subproblems’ software are projections of the behaviour of the completed software; the domain behaviours which the subproblems evoke are projections of the domain behaviours in the completed system; and the requirements of the subproblems, when fleshed out by the subproblem analysis, are projections of the functional requirement of the whole system.

There is a little paradox here. The problem frames approach aims to be firmly anchored in the physical problem world: the software in each subproblem must evoke a physical system behaviour that exemplifies the subproblem’s operational principle. Yet the software parts of these small systems are not expected to fit together snugly as subsystems of the completed software. The source of the paradox is the pursuit of simplicity and understanding in problem structure and analysis, deferring considerations of software architecture—which are concerned with the structure of the solution rather than the problem. A realistic computer-based system can be understood from many points of view and dissected and structured in many dimensions. For example, in a particular business system development it may be clear that the eventual implementation will be based on a three-tier architecture: the architectural view then has the three parts *ClientPresentation*, *BusinessLogic*, and *ServerDatabase*. The architecture of an embedded system may have four parts: *AcceptStimuli*; *ProcessStimuli*; *ControlOutputs*; *ManageDisplay*. These may be excellent structures for the software; but they are likely to be an obstacle—not an aid—to understanding the problems that the systems are intended to solve. From the point of view of problem analysis, they separate what should be brought together, and bring together what should be separated. Useful problem structures will be quite different. A simple correspondence between problem and solution structures is desirable, but may often be unachievable without doing violence to one of the two. The malleability—even fluidity—of software allows a rich repertoire of transformations: when the time comes the old wine of the problem-structured analyses can be carefully poured into the new bottles of the chosen implementation architecture.

4.4 Two Sources of Complexity

Each subproblem to be identified in a problem decomposition will correspond broadly to an identifiable useful functionality of the whole system: for example, in the library system *BookLending* and *MembershipManagement* may be identified as two subproblems. Decomposition to a much finer granularity will not increase understandability, because the resulting fragments of functionality, when considered individually, will have no intelligible purpose or operational principle. The point is readily illustrated by the analysis of a large finite-state machine. Understanding may be achievable by factoring into smaller quotient machines, or by identifying nearly decomposable regions. But decomposition into individual state transitions will hinder understanding, not help it. A single transition arc, considered individually, can have no intelligible purpose.

The complexity of any subproblem in a realistic system arises from contributions from two sources: from the intrinsic complexity of the subproblem function itself; and from the modification—perhaps even the distortion—of the function due to its interactions with other functions. The *BookLending* subproblem must deal with the basic events in which members borrow and return books, renew loans when they want to keep the book for longer than the standard loan period, reserve a book that is currently out on loan, cancel a reservation, and so on. In addition the subproblem must deal with such possibilities as the loss, theft or destruction of books. The *MembershipManagement* subproblem must handle initiating, renewing and resigning membership, payment of subscriptions, changes in members' circumstances—such as bankruptcy, emigration, change of name, prolonged illness—and so on.

Each of these subproblems, then, has its own intrinsic complexity. Additionally, the subproblems interact because books are to be borrowed only by members. A further potential contribution to the complexity of both subproblems is therefore the need to handle this interaction. Can a book be lent if the borrower's membership is due to expire within the standard loan period? Can a member be permitted to resign while still holding a borrowed book? What must happen if a member's name changes while a book is reserved but not yet borrowed?

4.5 Top-Down Decomposition, Bottom-Up Recombination

In the proposed approach to problem analysis, the two sources of subproblem complexity are separated. The decomposition into subproblems is carried out top-down. Each identified subproblem is regarded as defining a small closed system, ignoring its interactions with other subproblems of the system. For example, the *BookLending* subproblem may ignore the interaction with the *MembershipManagement* subproblem by assuming a context in which memberships, and the properties of individual members, are constant. The *recombination* of the subproblems, after their respective analyses, is carried out bottom-up. In the library system, the recombination will address the complexities arising from the interactions between the book lending and membership management processes.

Deferring the subproblem interactions in this way allows the intrinsic complexities of each subproblem to be studied and sufficiently well understood before the interaction complexities are addressed. Recognising the need for recombination as a distinct development task has implications for the whole development process. Because the subproblems identified in decomposition are oversimplified, some of the analysis may need to be reconsidered and possibly modified. By design, therefore, the development process is non-monotonic.

The decomposition is also unconventional in two other ways. First, because it ignores the interactions of the identified subproblems, the decomposition does not formally specify any relationship among them. In particular, if a problem *P* is decomposed into two subproblems *S1* and *S2*, the decomposition says nothing about *P* itself except that *S1* and *S2* will—eventually—contribute to its solution.[2] Second, the decomposition is not exhaustive: it does not exhaust the subproblems that will eventually make up the whole problem. The task of recombining the subproblems will, in general, reveal the existence of additional subproblems concerned with the analysis and management of interactions among the subproblems to be recombined.

Naturally there are opportunities and motivations for modifying this process; and judgment must be exercised in deciding how far the analysis of each subproblem should be carried before its combination with its siblings can be addressed. These judgments are made in the light of the general principle that the parts of a whole must be both identified and well understood before they can be recombined. The seventeenth-century clockmaker could not begin the design work for the first pendulum clock by designing the clock's frame. It was necessary first to design the parts—the weight, the pendulum and its suspension, the gear train, the escapement and the hands—before the necessary dimensions and properties of the frame could be determined.

5. Requirement Decomposition and Instrumental Decomposition

So far, the discussion of decomposition here has emphasised the identification of subproblems concerned with distinct functions that the system is required to provide. Each subproblem's requirement is a separate projection of the whole requirement, and its problem world is a projection of the whole problem world. This may be called *requirement decomposition*. It differs from *instrumental decomposition,* in which the responsibility for satisfying one requirement projection is divided between two loosely communicating subproblems. The difference is explained in this section.

5.1 Requirement Decomposition

Figure 3 shows a sketch of a system to control a lift in a hotel.

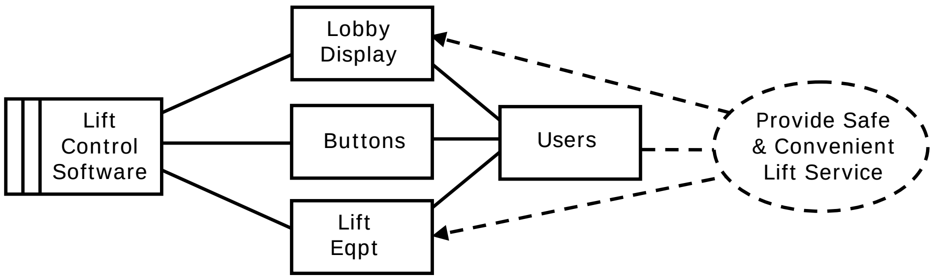

Figure 3. A Lift Control System

The stated system purpose is to 'provide safe and convenient lift service'; but this is not a simple purpose readily associated with a simply expressed and understood operational

[2] There is an exception to this statement, explained in the following section. In some cases a local variable of the software of *P* is identified by which *S1* and *S2* will communicate.

principle. On investigation, it appears, rather, to be some combination of at least three smaller purposes:

- to provide lift service in the usual sense, transporting users from floor to floor on request;
- to maintain safety by detecting equipment faults such as hoist motor failure, breakage of the hoist cable, or a stuck floor sensor, and taking appropriate action to avoid disaster; and
- to maintain a display in the hotel lobby, showing which floor the lift car is currently at and which floors have pending requests.

If this view holds, the decomposition must take the form of capturing these three identified subproblems. The first is shown in Figure 4.

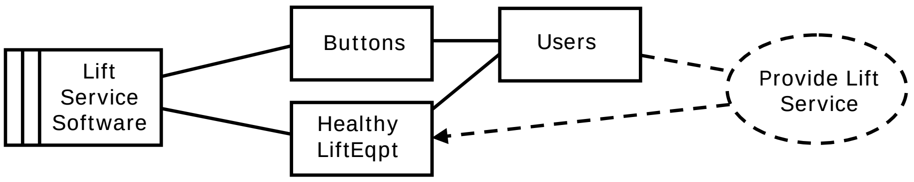

Figure 4. Lift Service Subproblem

In this projection, the lobby display is not relevant and has been omitted. Also, a local assumption has been made about the lift equipment problem domain. It is assumed that the equipment is healthy, in the sense that it is sufficiently free from malfunctions to provide the behaviours necessary to the lift service function. That is: when the software sets the hoist motor state to *upwards* and *on*, the lift car rises at the expected rate in the shaft; when the car reaches and leaves a floor the floor sensor state switches to *on* and *off* accordingly; and so on. This assumption, that fault-free behaviour is a given property of the lift equipment, is, of course, a strictly local assumption about the context of the subproblem.

The second subproblem is shown in Figure 5.

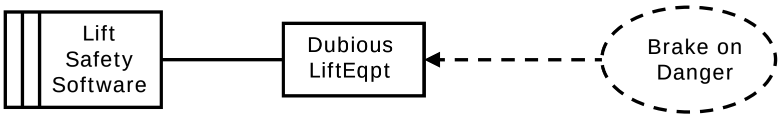

Figure 5. A Simple Lift Safety Subproblem

The purpose of this subproblem is to monitor the behaviour of the lift equipment in order to detect faults that could potentially endanger the lift users, and to take appropriate action when such a fault is detected. Here, of course, the local context does not assume healthy equipment. On the contrary, it assumes that the equipment is liable to faults. The given domain properties of the lift equipment are therefore those that allow faults—including incipient and impending faults—to occur: when the hoist motor state is *upwards* and *on*, the lift car may fail to rise at the expected rate in the shaft. To the extent that is possible and desirable, the given properties also allow faults to be detected, and the necessary precautionary action to be taken. Here we may suppose that the action to be taken is always the same: the hoist motor state is set to off and the emergency brake is applied, locking the lift car in the shaft so that it cannot fall freely.

5.2 Instrumental Decomposition

A different form of decomposition is *instrumental decomposition*. Requirement decomposition answers the question: What are the smaller and simpler purposes that contribute to the larger purpose of this problem? Instrumental decomposition answers the question: How can the intrinsic complexity of this problem's purpose be mastered? In an instrumental decomposition some internal interface of the undecomposed software—a structure of otherwise hidden phenomena—is exposed to serve as a medium of communication between the decomposed subproblems. For example, this interface may be a shared data structure, or a set of shared events. Unlike requirement decomposition, instrumental decomposition therefore has a substantial design aspect. In the lift control system, for example, the safety system may be decomposed as shown in Figure 6.

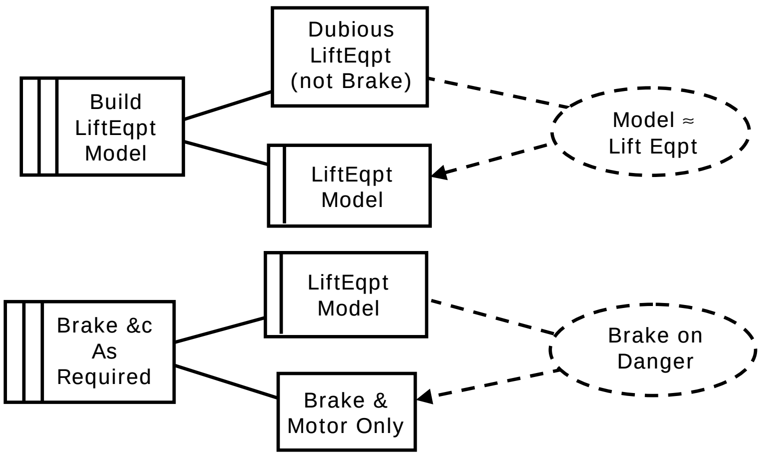

Figure 6. Instrumental Decomposition of a Safety Subproblem

In the safety subproblem the exposed interface is the LiftEqptModel. It is a shared data structure written by one of the subproblems and read by the other, functioning as a *model*—or software surrogate—of the lift equipment. The upper subproblem's software monitors the behaviour of the lift equipment (excluding the emergency brake, which is assumed to be fully reliable), and builds and maintains a dynamic representation in the LiftEqptModel of the equipment's current history and state, with particular focus on potential failures. The lower subproblem's software monitors the history and state of the model; it detects any dangerous situation recognisable in the model, and takes the necessary action. The primary design task arising from the decomposition is to design the data structure—probably as an instance of an abstract data type. The design will be determined by the information needed by the lower subproblem, and by the extent to which the upper subproblem's software can maintain a data structure from whose current value that information is directly available or can be derived.

The motive for this decomposition is to make a separation that is instrumental in satisfying the safety requirement. The separation is not in any way inherent in the safety requirement: it is a chosen instrument to simplify satisfaction of the requirement by separating two concerns. One concern is to monitor the behaviour of the lift equipment, continually checking the sensor states in relation to each other and to the current and recent motor settings, and maintaining a partially summarised record—that is, the model—from which existing, incipient and impending faults can be inferred. The other concern is to draw appropriate inferences from the model state as it changes, and to take any necessary action. The justification for this separation is the complexity of these two concerns.

6. SIMPLE OPERATIONAL PRINCIPLES

The discussion so far has merely asserted the simplicity of the chosen subproblems. If problem complexity is to be mastered by decomposition into simple subproblems, practical criteria are needed to distinguish the complex from the simple. The operational principle of a proposed subproblem provides the context for applying such criteria. When the subproblem purpose is elaborated or refined in the process of solving the subproblem—that is, specifying a software behaviour that can ensure satisfaction of that purpose—the operational principle provides a structure for the process. Traversing that structure, considering the subproblem requirement and the given properties of its domains, the developer may encounter points at which simplicity appears seriously compromised, and the choice of the subproblem in hand must be reconsidered. In this section some criteria of simplicity are mentioned and briefly discussed. These criteria are not disjoint. In the presence of complexity more than one criterion of simplicity is likely to be compromised.

6.1 One Level of Purpose

A simple operational principle has only *one level of purpose*. In many systems a cascading requirement stipulates a primary goal to be achieved, together with one or more levels of weaker goals to be achieved if the primary goal is unattainable. This is common in systems that must exhibit some degree of fault-tolerance. It may be recognised also in systems where the behaviour of a human participant in the problem world may fall short of what is normally expected. The criterion suggests that distinct levels of the requirement cascade should be treated in separate subproblems. For example, a bank loan customer may pass through several successive levels of delinquency by failing to meet the bank's loan conditions. Separating the treatment of the different levels will both clarify the conditions applicable to each level and expose the concerns to be addressed when the customer moves to a higher or lower level.

6.2 One Level of Abstraction

A simple operational principle is based on *one level of abstraction* of the phenomena of the problem domains. Suppose, for example, that in a system to control a car park the shared phenomena by which the software normally controls the raising and lowering of the barrier are abstracted as the events {*Raise* and *Lower*}; in some circumstances a finer-grained abstraction of the same phenomena—{*MotorOn*, *MotorOff*, *MotorUp*, *MotorDown*, *Open* and *Closed*}—may be appropriate. The two abstractions should not be applied in the service of the same operational principle; they should be applied in different subproblems.

To provide the more abstract interface {*Raise* and *Lower*} it will be necessary to translate between the two levels: the translation separates the car park management requirement from the detail of operating the barrier hardware. The translation is itself a distinct subproblem resulting from an instrumental decomposition. Both levels of abstraction will appear in the translation subproblem, but as phenomena of distinct problem domains. Only the fine-grained level appear as phenomena of the barrier domain; the more abstract {*Raise* and *Lower*} appear as phenomena of a domain introduced in the decomposition, behaving as a source of commands issued to the barrier. The stream of these commands is the structure of exposed phenomena of the undecomposed software.

6.3 Uniform Given Domain Properties

A simple operational principle assumes *uniform given properties* for each problem domain. For example, in the lift control example, the lift service subproblem must assume healthy operation of the lift equipment to the degree that is necessary to provide service. For the Lift Safety system, by contrast, the operational principle rests specifically on the potential for faults in the equipment, and on the relationships between internal equipment faults—such as a stuck sensor—and the evidence of that fault detectable at the interface with the software. This consideration alone indicates that the Lift Service and Lift Safety requirements are to be handled by distinct subproblems.

6.4 Synchronicity

Every system has a temporal dimension of execution. This temporal dimension may embrace asynchronous concurrent processes: it is then necessary to separate the concurrent processes into distinct subproblems. (In the turnstile problem the stream of coin insertions and the protocol by which each visitor negotiates the barrier may be regarded as concurrent processes.)

In the absence of true concurrency, a system may still perform functions of different periodicities. The tempi of these functions may be synchronised by nesting, just as the gear train of the pendulum clock has shafts rotating at different, but synchronised, speeds. In some cases, however, the tempi may be incompatible and cannot be nested—for example, if one behaviour is synchronised with calendar months and another with seven-day weeks, or one with lunar months and another with solar years. Incompatible tempi should be separated for treatment in distinct subproblems. An application of this idea to the spatial periodicity of iterative stream structures is found in the notion of a *structure clash* in the JSP design method [Jackson75] for sequential programs.

6.5 Uniform Domain Roles

In a simple operational principle each problem domain plays essentially *one role*. For example, in a system in which clerical workers edit documents, and management information is provided about the work they do, the workers are playing two roles. In one role they are the users in a document editing problem; in the other they are the subjects of an information display problem. The two roles should be separated into distinct subproblems.

6.6 Single Operational Phase

Many systems have distinct *phases* of operation. For example, in an avionics system the phases may be: pull-back from departure gate; taxiing; take-off; climbing; cruising; descent; landing; pull-in to arrival gate; and so on. Each phase is likely to have its own local assumptions of problem domain properties, and its own operational principle. To maintain simplicity of the operational principle, each phase should therefore be separated into distinct subproblems.

6.7 Completeness

A subproblem can embody a simple operational principle, and achieve an intelligible purpose which the developer can easily endorse, only if its problem world is in some sense closed and complete. First, in a closed problem world every state or event is regarded as controlled by some problem domain included in the problem world. This is an assumption of the local subproblem context: for example, in the Lift Safety subproblem shown in Figure 5 the changes in the motor state are regarded as spontaneous behaviour of the lift equipment, although in the Lift Service subproblem of Figure 3 they are regarded as controlled by the Lift Control Software. Second, the problem world must be complete in the same sense as a CSP process must be complete. That is: if the alphabet of a problem domain includes any events of a class, then all events of that class must be accounted for in the described domain properties.

In some software application areas it is common to find a style of requirement description by isolated fragments that makes this criterion hard to satisfy. For example, the following requirement appeared in a large specification of a chemical manufacturing plant [Harel09]: “When the temperature is maximum, the system should display a message on the screen, unless no operator is on the site except when T<60deg.” The apparent intention is to relate the dynamic behaviour of the temperature, the possible presence and absence of an operator, and the desired message display. Taken alone, this isolated statement cannot define an intelligible purpose for a subproblem: its satisfaction cannot be the purpose of a contrivance embodying an intelligible operational principle.

7. BOTTOM-UP RECOMBINATION

In identifying very simple subproblems, the decomposition oversimplifies by ignoring the eventual need for recombination. Each subproblem can then be analysed and understood in isolation. For example, when a subproblem associated with one mode or phase of the system is analysed, the local context of the subproblem treats that mode or phase as if it persisted over the whole operational life of the system. This approach flouts the dictum ascribed to Albert Einstein: “Everything should be as simple as possible, but no simpler,” the oversimplification being justified by the easier understanding of each subproblem in isolation. But in the end the dictum cannot be gainsaid, and the price is paid when the parts are recombined to constitute the whole. Recombination is itself to be regarded as a distinct task, in which the subproblems to be combined constitute the problem world.

7.1 Requirement Recombination

Subproblem recombination includes the task of bringing the subproblem requirements together in a coherent overall requirement.

Some recombinations may fall into the ambit of well-known techniques. For example, the combination of the two parts of the safety system—one building and the other using the LiftEqpt model—is a relatively straightforward case of managing access by a writer and a reader to a shared variable: the granularity of the interleaved accesses must take account of both syntactic and semantic properties of the model.

Sequential recombinations of parts associated with distinct phases are, in general, concerned with identifying and establishing a compatible state which can serve both as the termination state of the preceding phase and the initial state of the subsequent phase. It may be necessary to modify the phase subproblems to ensure this orderly switchover; or an additional subproblem may be identified that is responsible for reaching the compatible state before the switchover takes place.

Some recombinations demand the reconciliation of a conflict between the parts to be combined. For example, the lift service and lift safety requirements come into conflict when an equipment fault has been detected. The safety requirement demands that the motor be switched off; the service requirement demands that the motor continue to be available for sending the lift car to requested floors. In such a case it is necessary to concede priority to one part over another.

Where two subproblems have requirements that overlap in time, but make inconsistent local assumptions about problem domain properties, the inconsistency may be removed by suitable elaboration of one or both subproblems. For example, in the library system, the book lending subproblem may have assumed that membership is effectively static. In reality, the status of each member may change during the currency of one episode of lending. Both the lending and the membership subproblems have already been well understood and analysed in their simplified forms: in particular, the possible behaviours and event sequences have been elaborated for the lending subproblem, and the various member statuses and the transitions between them have been analysed in the membership subproblem. Viewing both subproblems as defining finite state machines, the developer can in principle construct their product machine. The states and events of the product machine can then be examined to identify impossible or undesirable events and transitions, and the software behaviours of the subproblems can be modified to eliminate them.

7.2 Software Recombination

A system comprising only one subproblem can be implemented by executing the subproblem's software. Where the system has more than one subproblem the subproblems' software must be recombined into a suitable architecture. In general, this recombination will involve transforming each subproblem's software. For example: the software of one subproblem may be dismembered and distributed in the text of the software of another subproblem; two similar but not identical models of a problem domain may be merged into one; a software behaviour including operations to read from an input stream of events or messages may be transformed into a procedure to be invoked as each element of the stream becomes available. [3]

Because the topic of this paper is problem analysis rather than software implementation, these transformations will not be further discussed here. Their significance for this paper is that the possibility of transformation releases the problem analysis from the obligation to fit the Procrustean bed of the eventual software architecture.

8. Formal Reasoning and Operational Principles

Polanyi stresses the distinction between science and mathematics on the one hand and operational principles of contrivances on the other, even when they are applied to the same physical objects:

> "The first thing to realize is that a knowledge of physics and chemistry would in itself not enable us to recognize a machine. Suppose you are faced with a problematic object and try to explore its nature by a meticulous physical or chemical analysis of all its parts. You may thus obtain a complete physico-chemical map of it. At what point would you discover that it is a machine (if it is one), and if so, how it operates? Never. For you cannot even put this question, let alone answer it, though you have all physics and chemistry at your finger-tips, unless you already know how machines work. Only if you know how clocks, typewriters, boats, telephones, cameras, etc. are constructed and operated, can you even enquire whether what you have in front of you is a clock, typewriter, boat, telephone, etc. The questions: 'Does the thing serve any purpose, and if so, what purpose, and how does it achieve it?' can be answered only by testing the object practically as a possible instance of known, or conceivable, machines. The physico-chemical topography of the object may in some cases serve as a clue to its technical interpretation, but by itself it would leave us completely in the dark in this respect."

In the development of computer-based systems, similarly, the role of science and mathematics is not to propose, or select, or establish operational principles of contrivances. Rather, it is to operate within the framework determined by the informally identified operational principles of the subproblems and their recombination in the system which is to embody them. Formal software development should be based on non-formal, clearly articulated, structures and operational principles.

[3] An example of this transformation is the *program inversion* scheme described in [Jackson75].

9. Concluding Remarks

The fundamental technique for mastering complexity is division of the complex object of study into simple parts. This technique has been known in principle since antiquity, and was compellingly reiterated [Descartes37] in one of Descartes' four principles:

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

but as Leibnitz pointed out [Leibnitz57]:

> "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."

The central theme of this paper is that the problem frames approach can help to reduce difficulty, and also to place formal and structural aspects of development in their proper relationship. Polanyi's notion of the operational principle of a machine or contrivance offers both justification and support for this approach, and clarifies the relationship between natural science and mathematics on one side, and what Polanyi calls "the logic of contriving" on the other. The applicability of this notion to software development has been argued here at length. Applicability in another field is evidenced by its enthusiastic adoption by the aeronautical engineer Walter Vincenti. He wrote [Vincenti93]:

> "Finally, the operational principle provides an important point of difference between technology and science—it originates outside the body of scientific knowledge and comes into being to serve some innately technological purpose. The laws of physics may be used to analyze such things as air foils, propellers, and rivets once their operational principle has been devised, and they may even help in devising it; they in no way, however, contain or by themselves imply the principle."

The broad structure proposed in this paper, in which the development problem is decomposed top-down and the decomposed parts subsequently recombined bottom-up, reflects the character of human understanding as a dynamic process. Just as developers gain in general understanding of their field during their individual working lifetimes, so in the same way they gain in specific understanding of each system to be developed during the progress of its development. The top-down initial identification and analysis of isolated simplified systems, followed by bottom-up recombination—demanding some adjustment and rework of what has already been done on the way down—can be seen as an instance of this learning process. Descartes' well-known principle of overcoming complexity by division into parts is accompanied [Descartes37] by three other principles. Perhaps the most apposite here is this:

> "...to conduct my thoughts in such order that, by commencing with objects the simplest and easiest to know, I might ascend by little and little, and, as it were, step by step, to the knowledge of the more complex; assigning in thought a certain order even to those objects which in their own nature do not stand in a relation of antecedence and sequence."

The "top-down, then bottom-up" sequence of problem analysis and understanding is valuable in itself, and is well supported by careful attention to the operational principle of each identified subproblem. For some readers, surely, this paper will have done no more than articulate a form of development process that they will recognise as their own usual practice.

Acknowledgments

This paper has been much improved by Daniel Jackson's comments and suggestions..

References

[Descartes37] Rene Descartes; *Discourse on the method of rightly conducting the reason, and seeking truth in the sciences*; Leyden, 1637; available as Project Gutenberg Etext #59 at http://www.gutenberg.org/etext/59 [accessed April 2009].

[Harel09] David Harel; *Statecharts in the Making: A Personal Account*; Comm ACM Volume 52 Number 3 pages 67-75, March 2009.

[Hoare81] C A R Hoare; *The Emperor's Old Clothes* (Turing Award Lecture); CACM 24,2, February 1981; reprinted in C A R Hoare and C B Jones, Essays in Computing Science, Prentice-Hall 1989.

[Jackson75] M A Jackson; *Principles of Program Design*; Academic Press, 1975.

[JacksonZave95] Michael Jackson and Pamela Zave; *Deriving Specifications from Requirements: An Example*; in Proceedings of the 17th International Conference On Software Engineering, pages 15-24, ACM and IEEE CS Press, 1995.

[Jackson01] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2001.

[Leibnitz57] G W Leibnitz; Philosophical Writings (Die philosophischen Schriften) Volume IV, ed C I Gerhardt, p331, 1857-1890.

[Polanyi58] Michael Polanyi; *Personal Knowledge: Towards a Post-Critical Philosophy*; Routledge and Kegan Paul, London, 1958, and University of Chicago Press, 1974.

[Vincenti93] Walter G Vincenti; *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History;* The Johns Hopkins University Press, Baltimore, paperback edition, 1993.