Proceedings of the IEEE Second International Symposium on Requirements Engineering, pp2-8; ACM Press, 1995.

Problems & Requirements

Michael Jackson
101 Hamilton Terrace, London NW8 9QX England
MAJ Consulting Ltd
& AT&T Bell Laboratories, Murray Hill, NJ, USA

Abstract

Requirements, specifications, and programs are distinguished by the phenomena they concern. Requirements are about phenomena of the application domain, and describe properties of the domain that the machine is required to bring about and maintain. The application domain is informal, and serious difficulties are encountered both in describing it and in reasoning about it. Requirements are complex, so they must be decomposed. Decomposition is based on the recognition of simple sub-problems, characterised by problem frames.

Keywords: Requirements, specifications, domain knowledge, problem frames

1 Why We Need Requirements

Software development aspires to be regarded as an engineering discipline. This aspiration can be justified on two grounds. First, the end product of software development is a useful machine. Dijkstra[1] pointed out a change in our attitude to computers: “It used to be the program’s purpose to instruct our computers; it became the computer’s purpose to execute our programs.” We might add: “It became the computer’s purpose to embody the machines that we describe by our programs.” In software development we build machines by describing them: the medium is text, but the products are engineering products. Second, because the machines, and the useful purposes they serve, are complex, we must master the complexity by appropriate structuring and design of our descriptions. We must engineer not only the machines, but also the texts by which we describe them.

Among the descriptions we must make to construct a machine is often a carefully engineered description of the requirement that the machine is intended to satisfy. The need for such a description is not universal in software development, and not common in certain other branches of engineering. It arises from the highly particularised nature of much software development. The engineer designing a new model of a family saloon car is solving a standard problem and making only small perturbations to an established standard design. There is no need to consider whether the car should be able to carry a 5-ton load, or should be equipped with a crane, or should be capable of travelling over deep snowdrifts. But an engineer developing a bespoke software system for a particular customer must approach the problem in a more open-minded way. The first step must be to capture the particular customer’s requirements.

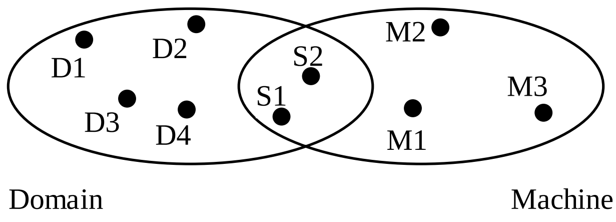
These requirements are not to be sought in a description of the machine to be built. They are to be sought in a description of the effects that the customer wants the machine to bring about in the world. An airline reservation system must bring about relationships among flights and passengers; a railway signalling system must bring about relationships among train movements, tracks, and timetables. A lift control system must bring about relationships among lift movements between floors, openings and closings of doors, illumination of indicator lights, and users’ requests for travel.

2 Requirements and Specifications

The requirement for a system, therefore, is to be expressed in terms of relationships among *domain phenomena*. When a user presses the up button at a floor, reasonably soon the lift should come to that floor and stop with the upwards arrow indicator illuminated. The doors should then open, and, after an interval long enough for the user to enter the lift, the doors should close. If on entering the lift the user presses the button for a higher floor, the lift should travel upwards and stop at that floor, possibly after stopping at others on the way.

All of the phenomena mentioned are observable in the application domain. The lift control system will be able to satisfy the requirement because — and only because — it shares some phenomena physically with the application domain. But these phenomena are not, in general, the phenomena of interest to the lift's users and the developer's customer. They are events and states at the interface between the machine and its environment. For example, there are event classes *MotorOn* and *MotorOff*, in which the winding motor of the lift is turned on and off. These events are shared by the motor and the machine, and occur only when initiated by the machine. From the point of view of an observer of the machine, each event is the sending of a signal on an output line; from the motor point of view, it is the connection or disconnection of electrical power. There is a set of states *SensorOn[i]* which hold when the sensor in the lift shaft at floor *i* is closed by the presence of the lift car. These states are shared by the lift domain and the machine; they are controlled by the domain. In the machine they appear as an array of Boolean values; in the lift they appear as the positions in space of the sensor levers.

The relevant sets of phenomena may be shown in a Venn diagram:



The phenomena {Di} are private to the domain; {Mi} are private to the machine; and {Si} are shared by both. The requirements are expressed as relationships over the domain phenomena {Di}∪{Si}. Call these requirements u. They will be satisfied by constructing a machine. The machine's external behaviour and properties will be described in a specification v, expressed as relationships over the shared phenomena {Si}. For example, when the machine detects that *Sensor[3]* is true when there is an outstanding request to visit floor 3, it will cause a *MotorOff* event.

A specification in this sense is a restricted kind of requirement. It is a requirement because it is a relationship over domain phenomena; it is restricted because it must be expressed solely in terms of domain phenomena that are shared with the machine. A program, of course, is scarcely ever a requirement: it is a relationship over the machine phenomena {Mi}∪{Si}.

3 The Role of Domain Properties

It is possible for the specification v to satisfy the requirement u because the domain possesses certain properties that are independent of the behaviour of the machine. When the motor is turned on, the lift starts to rise within 250 milliseconds; when it is turned off the lift stops moving; when the car arrives at a floor the sensor state changes from off to on. Let us call these domain properties $\mathcal{D}$.

It is these domain properties that allow the specification $\mathcal{S}$, restricted to shared phenomena, to satisfy the less restricted requirement $\mathcal{R}$. If the development has been correctly carried out, we have the entailment:

$$\mathcal{S},\ \mathcal{D} \vdash \mathcal{R}$$

That is, if the machine behaves as specified, and the domain has the properties we claim, then satisfaction of the requirement can be deduced.

For purposes of constructing, rather than validating, the specification $\mathcal{S}$, it can be viewed as a refinement of the requirement $\mathcal{R}$. In this refinement, the underlying domain properties $\mathcal{D}$ play the role that is played in program refinement by the semantics of the programming language — that is, by the underlying properties of the computer.

4 A General Problem Frame

This view of requirements and specifications is very general. But it is not vacuous. It distinguishes three parts of the problem: the application domain, the requirement, and the machine we will build. It also offers an element of method by indicating a relationship among three descriptions of interest: the requirement, the domain properties, and the machine specification. (If our scope included programming we would add the executable program text as a fourth description.)

There is a close similarity to the problem-solving ideas of the ancient Greek mathematicians expounded by Polya[2]. Polya identifies two problem classes: problems to prove, and problems to find or construct. Each class of problem is characterised by its principal parts and a solution task, which together may be called its *problem frame*. For example, the problem 'Given lengths *a*, *b*, and *c*, construct a triangle whose sides have those lengths' is a problem to find or construct. The principal parts of such a problem are:

> the *data* (here, the three lengths);
>
> the *unknown* (here, a triangle); and
>
> the *condition* (here, that the triangle's sides be of the three lengths).

The *solution task* is to construct the *unknown* so that it bears the relationship to the *data* described by the *condition*.

By contrast,the problem 'Prove that the angles at the base of an isosceles triangle are equal' is a problem to prove. The principal parts of such a problem are:

> the *hypothesis* (here, that the triangle is isosceles); and
>
> the conclusion (here, that the angles at its base are equal).

The *solution task* is to demonstrate that the *conclusion* follows from the *hypothesis*.

Software development problems are very similar to Polya's problems to find or construct. The principal parts are:

> the application domain;
>
> the machine; and
>
> the *requirement*.

The *solution task* is to construct the *machine* so that it ensures that the *requirement* holds in the *application domain.*

5 Heuristics for Solution

A problem frame makes it possible to talk usefully about solution methods for problems that fit the frame. Polya offers several heuristic recommendations. For a problem to prove:

consider what other *conclusions* follow from the *hypothesis*;

split the *conclusion* into parts;

and so on. For a problem to find or construct:

check that you are using all the *data*;

ask whether the condition is sufficient to determine the *unknown*;

and many others. Such a discussion of methods depends on being able to talk about the named principal parts of the problem.

For software development problems, we may offer in similar vein:

describe the *application domain* properties independently of the *requirement*;

describe the *requirement* separately;

look for additional relevant and useful properties of the *application domain*;

enumerate the phenomena shared by the *machin*e and the *application domain*;

describe the *machine* in a specification expressed in terms of shared phenomena;

and so on.

6 Difficulties of Informality – 1

Very often, the application domain is informal. It will be so whenever it is physically tangible, because the physical world is certainly informal at the level at which it can interact directly with a computer system. Even intangible parts of the application domain are likely to be informal, because they may be the product of human discourse or perception. For example, tax laws and negotiated wage agreements are parts of the application domain of any payroll system, and rules of visual elegance and balance are a part of any serious typesetting system such as TEX.

This informality presents an important challenge to software developers — to describe the application domain precisely, and to reason about it accurately and explicitly. Unfortunately, the challenge is easily declined. We may retreat into abstraction, preferring clean mathematical concepts to the messy phenomena of the real world. We may retreat to the boundary with the machine, because the shared phenomena are easier to deal with: being machine phenomena too, they must be already formalised by the well-specified behaviour of the computer. Or we may lay the burden of understanding the application domain on our customer's shoulders, insisting that the customer speak to us in well-defined terms. If twenty meanings of the word 'sale' are in use in the customer organisation, let the customer resolve that conflict. If no-one at the phone company knows exactly what is meant by a 'telephone call', we'll come back later when they've worked it out.

The challenge should not be declined. Formalising the informal is central to software development. The basic task here is to identify and exploit a set of phenomena that can act as 'ground terms' for our descriptions and reasoning. These *designated* phenomena must be events and states that can be described precisely enough to be reliably recognised in the application domain. In the presence of call-forwarding, conference calls, call waiting, and chat lines, it may be difficult to say how to recognise a telephone call. But it is easy enough to recognise when a telephone handset has been lifted ‘offhook’ or replaced ‘onhook’. An enthusiast who likes to dismantle and rebuild cars may find it difficult to say whether the car owned today is the same car as last week’s. But the licensing authority may find it quite easy to recognise whether it has the same engine block.

The point is that the application domain is often best approached in a reductionist spirit. The simplest and most distinctly observable phenomena form the soundest basis for description. More complex concepts can be build from them by appropriate use of definition. The distinction between designation of phenomena and definition of terms is fundamental to any serious attempt to describe the world.

7 Difficulties of Informality – 2

Not all the difficulties of informality are overcome by careful designation of domain phenomena. Because designation can never be perfect, there is always some residual error. Designated phenomena are to real phenomena as numbers in a fixed-length representation are to real numbers. And, as in real arithmetic, the errors can falsify apparently true assertions and vitiate the results of sound reasoning.

The point is clearly illustrated by a well-known mishap that occurred to an aeroplane landing on an airport runway. The plane’s control system was required to ensure that it is possible for the pilot to engage reverse thrust (PRV is true) if, and only if, the plane has already touched down and is moving along the runway (ONR is true):

$$\mathcal{R} : \text{PRV} \equiv \text{ONR}.$$

PRV is a phenomenon shared with the machine: the machine directly controls the interlock that disables reverse thrust. The phenomenon ONR is not shared with the machine. However, if (and only if) the plane has touched down and is moving along the runway (ONR is true), the wheels are turning fast — that is, at at least 5 revolutions per second (WLS is true). And if (and only if) the wheels are turning fast, pulses are being generated at a rate proportional to their speed of rotation, that is, at at least 20 pulses per second (PUL is true). So:

$$\mathcal{D}: \quad \text{ONR} \equiv \text{WLS; and} \\ \quad\quad \text{WLS} \equiv \text{PUL}.$$

The wheel pulses are phenomena shared with the machine. The derived specification was

$$\mathcal{S}: \quad \text{PRV} \equiv \text{PUL}$$

which is clearly based on correct reasoning:

$$\mathcal{S}, \mathcal{D} \vdash \mathcal{R}.$$

Unfortunately, there was a flaw in the domain description g. The plane was landing in heavy rain, and the runway was covered in water. The wheels were aquaplaning instead of turning, so the assertion

$$\text{ONR} \equiv \text{WLS}$$

was false in this situation. In the absence of wheel pulses, the control system prevented the pilot from engaging reverse thrust, and the plane overshot the runway.

8 Specific Problem Orientation

This view of requirements and specifications is very general, so inevitably it is weak. It is a principle of methodology that the power of a method is inversely proportional to its generality. The heuristics given earlier — ‘look for additional relevant and useful properties of the *application* *domain'* — were limited in power because they made no distinction between the structure and characteristics of one application domain and another. The account given of software development problems applied as well to a compiler as to an airline reservation system. To be powerful, a method must exploit the problem's features very minutely. Because problem features vary widely, we need a repertoire of methods, each suitable for problems of a particular class. Even in the realm of small mathematical problems the Greeks recognised two problem classes. In software development we must surely recognise many more than that.

A problem class is characterised by its problem frame. The principal parts of a problem frame are furnished by the parts of the world — the machine and the application domain — that form the problem context, and by the relationships among them. One approach to distinguishing different classes of problem is to distinguish different characteristics and structures in the machine. This may be called a *solution-oriented* approach, since the machine constitutes the solution rather than the problem itself. It may seem perverse to adopt a solution-oriented approach, but it is common. In fact, many of the traditional techniques of analysis can be seen as solution-oriented: problem analysis may be their ambition, but their technique is to structure the machine.

A problem-oriented approach seeks to distinguish different characteristics and structures in the application domain. Consider, for example, the restricted class of problem whose application domain can be described as follows. There is a serial stream of *operation requests*; these are requests from users of the system to perform operations on *workpieces*. The *workpieces* may be, for example, texts held in the machine. The machine is viewed as a *tool*, its purpose being to fashion the *workpieces* just as a metal workpiece is fashioned on a lathe or milling machine. The requirement is a desired relationship between the *operation requests* and the states of the *workpieces*. We might call this relationship the *operation properties*.

This frame would fit a problem like the construction of an extremely simple editor. The principal parts of the problem frame are:

the workpieces;

the operation requests;

the tool; and

the operation properties.

The solution task is to construct the *tool* so that it operates on the *workpieces* in response to *operation requests*, in accordance with the *operation properties*.

A simple model-oriented method based on abstract data types would be suitable for solving such a problem. The *workpieces* are instances of the type, and the *operation requests* invoke operations of the type. We can be confident that such a method would be suitable because the problem frame, stated in full, imposes further constraints on its parts. The *workpieces* must be intangible, physically realised only within the *tool*; dynamic — they can change state; and inert — they change state only as a result of externally controlled events. The *operation requests* must be viewed as an unstructured autonomous stream of request events. There is no notion of users as individuals, so there is no way, for example, of providing for user preferences in text styles. The stream is autonomous: requests are presumed valid and can not be rejected by the *tool*. The *tool* is reactive, rather than active. It does nothing except respond to requests; it has no behaviour other than its ReadRequest–PerformOperation loop.

9 More Simple Problem Classes

It is the simplicity — the unrealistic simplicity — of the Workpieces problem frame that gives it its power. Because the principal parts are tightly constrained it is possible to apply the associated method very easily and directly. We have, as it were, moved up only one level from a classroom problem such as ‘Write a model-based specification of a double-ended queue’, or ‘Extend the stack specification to accept *top* and *pop* operations when empty, and to respond appropriately’. The complications that plague realistic developments have been purged.

There are many other simple problem frames, although not many have been explicitly identified and described. Among them are the Simple Control frame, and the Simple IS frame.

The Simple Control frame is really a carefully purged version of the general frame discussed earlier. Its principal parts are:

the controlled domain;

the desired behaviour; and

the *controller*.

The *solution task* is to construct the *controller* — that is, the machine — so that it enforces the *desired behaviour* in the *controlled domain*. The *controlled domain* must be directly connected by shared phenomena to the *controller*; it must be dynamic, and partly autonomous and partly reactive. The *desired behaviour* is a relationship among the states and events of the *controlled domain*.

A method associated with the Simple Control frame, albeit with a far more elaborate version of the frame, is discussed by Parnas and Madey[3]. Appropriately, they stress the distinction between their mathematical relation *NAT*, which captures the natural properties of the *controlled domain*, and their relation *REQ*, which captures the *desired behaviour*.

The Simple IS frame underlies a number of development methods, including JSD[4]. Its principal parts are:

the *real world*, about which information is to be provided;

the information requests;

the information outputs;

the system; and

the information function.

The *solution task* is to construct the *system* so that it produces the *information outputs* in response to the *information requests*; the outputs must contain information related to the *real world* as specified in the *information function*. The *real world* must be dynamic and autonomous, its behaviour expressible as a set of regular expressions over events. The *information requests*, like the *operation requests* of the Workpieces frame, form an unstructured autonomous stream of request events. No distinction is made between a request from one user and a request from another.

10 Complexity and Decomposition

Problem frame variety is the key to mastering complexity. A complex requirement must somehow be broken down into simpler, more manageable, partial requirements. This process can be guided by treating it as the decomposition of complex problems into simple problems. A simple problem is one that we know how to solve: that means one for which we have a close-fitting problem frame and an effective associated method.

In identifying or formulating effective methods and problem frames, the purpose is therefore to ensure that the frame is so tightly defined that any problem that fits it must be soluble by systematic application of the associated method. This purpose rules out solution frames, because their suitability to a particular problem — or lack of suitability — emerges only very indirectly and very late in the development process. It also rules out very loose problem frames, because they fit no problem tightly enough to give confidence that it can be solved by applying the associated method. That is why top-down methods are of little value, and why Fred Brooks'[5] adjuration — 'Plan to throw one away' — rings so true in the ears of top-down developers.

Problem decomposition depends on the developer's ability to *recognise* the simple problems within the complex problem. This recognition is based on analysis of the application domain by fitting parts of the domain and parts of the overall requirement into the principal parts of known problem frames.

Imagine, for example, the development of an editor in which certain operations are forbidden to certain users. This can obviously be viewed as a complex problem consisting of a Workpieces problem combined with a Simple Control problem. Neither frame alone can accommodate the complete problem: the Workpieces frame has no provision for distinguishing one user from another or for constraining occurrences of *operation requests*; and the Simple Control frame has no provision either for the *workpieces* themselves or for the *operation properties* of the operations to be performed on them.

The decomposition into the two subproblems can be easily understood by supposing one of them to be already solved. Suppose that the Workpieces problem has been solved, and the resulting system is in operation. Then the Simple Control problem is concerned with an application domain in which there are people using the editor that has been built. The people, the editor, and the texts being edited together form the *controlled domain*. The *desired behaviour* is that certain people do not use the editor in certain ways. The *controller* is, in principle, another machine that we must build to ensure that behaviour is as desired.

11 Some Consequences

The two simple problems are combined in a parallel, rather than hierarchical, structure. Phenomena are shared between principal parts of different frames, just as they are shared between different subdomains of the application domain and between the application domain and the machine. The events in the *operation requests* part of the Workpieces frame are also events in the *controlled domain* of the Simple Control frame. Some phenomena are private to one frame. The users of the editor, as individuals to be distinguished one from another, are private to the Simple Control Frame: in the Workpieces frame the originators of the operation request events are, as it were, anonymous.

This sharing and privacy has important consequences for description technique, and for the structures within which individual descriptions must be fitted. Large-scale parallelism is a crucial need in requirements engineering: further descriptions can always be superimposed on those already made. In broad terms, this point is widely recognised: for example, additional ViewPoints[6] can always be added in parallel to those already specified. But in specific terms the richness of the mutual constraint of two parallel problem frames raises issues that have not been widely addressed.

We may illustrate this richness by two examples. First, a class of shared phenomena may not be a class of explicit interest in either of the parallel frames taken in isolation. Suppose that in frame A the event classes EA1, EA2, and EA3 are identified. This classification scheme is appropriate to the concerns of the frame. In the parallel frame, B, the event classes EB1, EB2, EB3, and EB4 are identified for analogous reasons. Then a class of shared events ES may be such that:

$$ES \subset (EA1 \cup EA2)\text{; and}$$

$$ES \subset (EB1 \cup EB3 \cup EB4).$$

Its members appear in both frames, but are drawn from a union of subclasses in each frame.

Second, the control properties of events may be different in different frames. An event that is controlled by the machine in one frame may appear as an autonomous event of the application domain in another frame. Consider, for example, an event in which a user requests an operation on a *workpiece* in the Workpieces frame. This is a phenomenon shared by the *operation requests* and the *tool* principal parts of the frame. It is controlled by the users who issue the *operation requests*. In the Simple Control frame the same event is seen as an event of the *controlled domain*, this domain comprising both the *operation requests* and the *tool* of the Workpieces frame. It is an autonomous event in the *controlled domain*, but if it is forbidden for the issuing user then the *controller* in the Simple Control frame must inhibit it, directly or indirectly. Since the human user can not be prevented from making the request, the *tool* must be elaborated. For example, before performing any requested operation it must check with the *controller* that the operation is permissible. (It is worth remarking that any message to the user explaining why the operation has been inhibited must naturally come from the *controller* in the Simple Control frame, not from the *tool* in the Workpieces frame. The fact that they are almost certainly implemented on the same computer increases, rather than diminishes, the importance of the separation of concerns.)

The richness and fluidity of shared phenomena in parallel problem frames should not be taken to imply that requirements description must necessarily be informal. On the contrary. The richer and more complex the phenomena and relationships to be described, the more we need to formalise our thinking about them. But the formal descriptions must be expressed in languages, and related within structures, that do justice to the complexity of their mutual constraints and the multi-faceted nature of the phenomena and relationships they describe.

Acknowledgement

Pamela Zave read an earlier draft of this paper and made many helpful comments.

References

[1] Edsger W Dijkstra; A Discipline of Programming; Prentice-Hall, 1976.

[2] G Polya; How To Solve It; Princeton University Press, 2nd Edition, 1957.

[3] D L Parnas and J Madey; Functional Documentation for Computer Systems Engineering (Vn 2); CRL Report 237, McMaster University, 1991.

[4] M A Jackson; System Development; Prentice-Hall International, 1983.

[5] Frederick P Brooks, Jr; The Mythical Man-Month: Essays on Software Engineering; Addison-Wesley,1975.

[6] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein; A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification; IEEE Transactions on Software Engineering, Volume 20 Number 10 pages 760-773, October 1994.