# M253 Resource Sheet

## *How to decompose a problem*

### 1 Overview

The purpose of this Resource Sheet is to introduce you to several approaches to the task of breaking down a large problem into a collection of more manageable sub-problems. We do not intend that you should attempt to follow in detail any one of the suggested approaches as you attempt the decomposition of the problem posed in your project scenario. What we hope is that the discussion and examples will give you some ideas that will help you in your task.

### 2 Getting to grips with a problem

In focusing on problem identification you must steer a course between being too general and imprecise about the world in which the problem exists, and being too specific and precise about potential solutions before you understand the problem. Initially you need to focus on *what* the system will do rather than *how* it will do it, and part of the solution is to think about *where* it will do it. The problem is located in the real world; the solution is located in the computer and its associated software.

Another aspect of identifying the problem to be solved is to ensure that you consider it from the point of view of the users of your system. The emphasis is not just about how the users will interact with the system interface but also about how the features that the system provides will support them in the activities which they have to undertake in the real world.

If you do not identify the right problem initially then you may build software that correctly solves the wrong problem. Justified complaints from users of the system will then be not so much about the failures of the hardware or the software but about the unexpected and unwanted side effects that their use produces in the real world, due to the solution not matching the real problem.

This is emphasised in the following quotation from Michael Jackson's book *Problem Frames: Analysing and Structuring Software Development Problem*.

> 'It is important to focus directly on a problem, not just going straight to the design of a solution. The computer and its software are the solution; the problem is in the world outside the computer. In spite of good intentions, you can easily confuse the problem with its solution [...] This book is about the analysis of problems, not about solutions.'
>
> Jackson (2001) p.1

Another interesting quotation from the same work makes a related point.

> 'For many people in software development the computer remains far more interesting than the problem world.'
>
> Jackson (2001) p.10

If you are interested in programming, it is all too easy to rush into coding the solution before you have actually established what the real problem is.

Only when you have reached the situation where you think that you have understood the overall real world situation and the problem that you are being asked to solve, and you have elicited the real needs of all the stakeholders that you have identified in the problem domain, can you begin to think about how you might proceed to form and describe possible solutions to the problem.

## 3 Breaking down the problem

Typically your overall problem will be fairly complex and your overall solution will be equally complex. You need to think of ways in which you can manage that complexity. The traditional way of doing this is to decompose the original problem into a collection of simpler sub-problems, each of which can be handled — and hopefully solved more simply — on its own. At some later stage, by a reverse process of composition, you can then combine the various solutions to your sub-problems into a solution to the overall problem.

A clear justification for this approach, taken from the mathematician George Polya's classic book *How to Solve It*, states that:

> 'Decomposing and recombining are important operations of the mind. You examine an object that touches your interest or challenges your curiosity: a house you intend to rent, an important but cryptic telegram, any object whose purpose and origin puzzle you, or any problem you intend to solve. You have an impression of the object as a whole but this impression, possibly, is not definite enough. A detail strikes you, and you focus your attention upon it. Then, you concentrate upon another detail; then, again, upon another. Various combinations of details may present themselves and after a while you again consider the object as a whole but you now see it differently. You decompose the whole into its parts, and you recombine the parts into a more or less different whole.
>
> If you go into too much detail you may lose yourself in details. Too many or too minute particulars are a burden on the mind. They may prevent you from giving sufficient attention to the main point, or even from seeing the main point at all. Think of the man who cannot see the wood for the trees. Of course we do not wish to waste our time with unnecessary detail and we should reserve our effort for the essential. The difficulty is that we cannot say beforehand which details will turn out ultimately as necessary and which will not.
>
> Therefore let us, first of all, understand the problem as a whole. Having understood the problem, we shall be able to judge which particular points may be the most essential. Having examined one or two essential points we shall be in a better position to judge which further details might deserve closer examination. Let us go into detail and decompose the problem gradually but not further than we need to... it is a very foolish and bad habit with some [students] to start working at details before having understood the problem as a whole."
>
> Polya (1957) pp.75–76

In order to decompose your problem you need some guidance as to the sort of criteria you should use for identifying suitable sub-problems and for evaluating the degree to which these sub-problems are independent, or to which they are inter-related.

Ultimately, what you want to achieve is a decomposition of your original problem into a collection of simpler sub-problems that cover all aspects of the original problem, and that have clearly understood interactions with each other.

## 4 Problem frames

In the wider world of problem solving you might expect there to be classes of sub-problem that reappear in a variety of contexts. Indeed, for some such classes, you might expect ready-made solutions out there that can be picked up and (re)used in your composition process with little or no modification.

Amongst the authors who have spent a lot of time thinking about such issues is Michael Jackson, whose recent book *Problem Frames: Analysing and Structuring Software Development Problems*, contains some interesting conclusions and was the source of some of our earlier quotes.

Jackson begins by pointing out that the field of software development is much less specialised than more traditional engineering disciplines, and that the individual developer, and the individual development project, introduce a lot more variety. You rarely find yourself solving an immediately recognisable and well understood problem at the top level. Most real-life problems are, in any case, too big and complex to handle at a single level. You need to structure the overall problem in terms of interacting sub-problems. Jackson claims that this task can be made much easier by the use of problem frames.

What Jackson means by a **problem frame** is an attempt to identify and describe a recurring situation, to define a simple problem class. When attempting to identify suitable sub-problems, system developers can use his existing list of problem frames as a guide, looking for aspects of the overall problem that might fit a given frame. Then, when a sub-problem is recognised as an instance of a specific problem frame, system developers can draw on the experience associated with their previous use of that frame.

The use of problem frames is not associated with any particular software development methodology. If anything, it is intended as an antidote to the use of methodologies at too early a stage in the software development process. Most methodologies are strongly solution-oriented and tend to assume that the problem to be solved is well understood. The use of problem frames is proposed as a precursor, which will assist with the process of analysing and structuring the initial problem prior to looking for a solution.

Jackson makes a distinction between the use of the word ‘system’ to describe the whole combination of the world and the computer together as opposed to just the hardware and software at the centre. In this context he comments that you are attempting to describe the wider system (where the problem resides) rather than the narrower system (where the solution will reside). Jackson reserves the word ‘machine’ for situations in which he wants to talk about this narrower system, the computer and its software.

Jackson also makes a distinction between analytic and analogic models; in the first of these, ‘modelling’ implies that you are describing the system in the outside world, whereas in the second "modelling" implies that you are describing the system inside the computer. As far as you are concerned, at this stage of your project you are concentrating on the analytic activity of modelling what happens in your real-world system rather than how you will eventually represent the system in the computer.

There are five basic problem classes captured in Jackson's problem frames, together with a number of flavours or variants on these in order to accommodate a realistic range of problems. Each frame captures a class, and provides a frame diagram and associated frame concerns and development descriptions. The names that Jackson gives to the five basic frames are: Required Behaviour, Commanded Behaviour, Information Display, Simple Workpiece and Transformation.

**Note.** What follows are Jackson's brief descriptions of the problem class that each of these frames addresses, but we do not take the discussion into any further detail here. We are only introducing problem frames to provide you with *one possible way of thinking* about the different categories of sub-problems that you might look for in decomposing your own problems. We are *not* proposing that you go for the full Jackson treatment in the context of this project, although you might find it both interesting and useful to read the book at some later date.

### *4.1 The Required Behaviour frame*

> 'is intended to capture the idea that there is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control.'
>
> Jackson (2001) p.85

A simple example of this is a controller for a set of one-way lights to manage the traffic flow at some road works.

### *4.2 The Commanded Behaviour frame*

> 'is intended to capture the idea that there is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.'
>
> Jackson (2001) p.89

A simple example of this is the controller for your video player.

### *4.3 The Information Display frame*

> 'is intended to capture the idea that there is some part of the physical world about whose states and behaviour certain information is continually needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.'
>
> Jackson (2001) p.92

A simple example of this is the speed and distance-travelled information provided on a car dashboard display.

### *4.4 The Simple Workpiece frame*

> 'is intended to capture the idea that a tool is needed to allow a user to create and edit a certain class of computer processable text, or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can act as this tool.'
>
> Jackson (2001) p.96

As a simple example of this you could have a tool to create and update information on an individual's wine purchases and tasting notes.

### *4.5 The Transformation frame*

> 'is intended to capture the idea that there are some given computer readable input(file)s whose data must be transformed to give certain required output(file)s. The output data must be in a particular format, and it must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs.'
>
> Jackson (2001) p.99

As a simple example of this you could have a program to analyse the data relating to an individual's weekly supermarket shopping bills and to identify purchasing patterns, so that a set of special vouchers can be generated to encourage them to buy more of specific items.

## 5 Heuristics

The benefits of a good decomposition of a problem are that, in addition to helping you to understand the problem, it should help you to describe/document the problem more clearly and also help you in your attempts to solve the problem. However, there are no hard-and-fast rules that can be laid down for how to achieve a good decomposition or even how to recognise that you have achieved a good one. It is important to realise that, apart from a few relatively trivial problems, there is likely to be a wide range of possible decompositions, many of which are equally ‘good’, rather than a unique ‘best’ decomposition.

In this context, Jackson comments that:

> ‘Problem decomposition is not an exact science. But it can exploit some useful heuristics, and it can be reasonably systematic.’
>
> Jackson (2001) p.269

Possible heuristics include the following.

- Identify the core problem – there is often one obvious central need that the system is intended to handle, so go for this first and work outwards from it.
- Identify ancillary problems – around the core there are many sub-problems, often in the form of information sub-problems related to the core sub-problem.
- Use standard decompositions of sub-problems – some of the problem frames themselves are naturally of a composite nature, and hence lead to the identification of further sub-problems.
- Identify common concerns and difficulties – when the analysis of identified sub-problems shows that two or more of them have common issues associated with them this may indicate the existence of further sub-problems to handle those issues.
- Look for sub-problems with different tempi – if there are activities that need to take place over very different time scales then these should be treated as separate sub-problems.
- Look for sub-problems with different moods – it may be appropriate to separate sub-problems relating to what the customer ideally wants from the system from sub-problems relating to what the customer must have from the system.
- Look for residual complexity – any sub-problem which is still relatively complex should be broken down into further sub-problems.
- Investigate any need to model users – if different groups of users need to access (any specific parts of) the system then treat each user group as a separate sub-problem.

## 6 Assessing the decomposition

When you come to the end of your decomposition activity, how should you judge the success of your efforts? You should have a collection of sub-problems that are each smaller and simpler than the original problem. They should each be a complete problem in their own right, in the sense that they do not depend on any of the other sub-problems for their solution. They should also be complete as a set, in the sense that between them they cover all the essential issues raised by the original problem. Insofar as the sub-problems need to interact with each other you should have a clear description of the nature and extent of their interactions.

Jackson (2001, p.65) points out that at the initial stage of decomposing your problem into sub-problems you usually find yourself creating several potentially overlapping 'projections' of the original problem, that are different **views** of that problem, rather than 'partitioning' your problem into totally independent sub-problems. These projections may well share data and need to react to the same external or internal events affecting the system.

You can see the difference in Figure 1. In the left-hand part of the diagram the problem is partitioned into five non-overlapping areas, with the implication that no element of the problem appears in more than one of these areas. In contrast, in the right-hand part of the diagram, each of the projections overlaps with one or more of the other projections, so that some elements are shared between the different views of the system represented by the different projections.

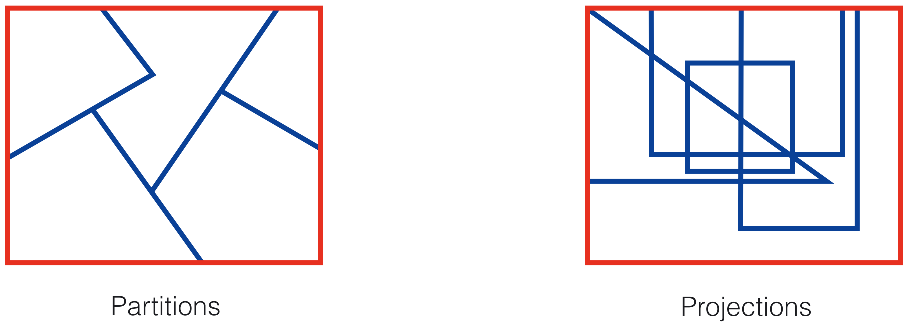

Figure 1 Partitions versus projections

## 7 Needs and features

A somewhat different emphasis than the problem frames approach, and one perhaps more familiar in traditional requirements engineering, is to concentrate on the issue of what the users, or more generally the stakeholders, want from the system that is to be developed. The Rational Software White Paper 'Features, use cases, requirements, oh my!' by Dean Leffingwell provides a very simple, high-level overview of some of the more important issues that arise in this context.

Leffingwell (2000, p.2) starts by reminding us that the purpose of the requirements definition phase of system development is to answer the very important, fundamental question: 'What, exactly, is this system supposed to do?'

He goes on to distinguish between the real needs of the stakeholders, as seen in the problem domain, and the features of a system that will be able to meet those needs, as provided in the solution domain.

Leffingwell (2000, p.2) defines a **stakeholder need** as 'a reflection of the business, personal or operational problem (or opportunity) that must be addressed to justify consideration, purchase or use of a new system.'

Leffingwell (2000, p.4) defines a **feature** as 'a service that the system provides to fulfil one or more stakeholder needs.'

He emphasises the fact that these features are *not* to be regarded as just a refinement of the stakeholders' needs. Rather, they are a direct response to the problems indicated by the stakeholders, and as such they provide a top-level solution to the problem. These features are described in natural language so that the stakeholders can easily understand what the proposed system is going to do, and are only concerned with communicating intent, with no hint of how the system might deliver them.

Leffingwell indicates that you should be able to describe a system by defining between 25 and 50 features that characterise its behaviour. More than this suggests an inadequate level of feature abstraction, or possibly an overlarge system that needs to be divided into several smaller pieces.

He also emphasises the way in which consideration of use cases can be beneficial in the process of defining system behaviour. Leffingwell (2000, p.4) defines a **use case** as 'the description of a sequence of actions, performed by a system, which yields a result of value to the user.' We could say that use cases describe how users and the system work together to realise the identified features. Typically there will be several use cases to indicate how a particular feature is to be implemented. Consideration and elaboration of these use cases moves you closer to your solution in behavioural terms, although it still keeps you away from the consideration of software requirements.

## 8 Summary

In this Resource Sheet we have looked at the overall need to break a problem down into a collection of sub problems, and have considered two rather different approaches to meeting this need. We are not expecting you to follow either of these approaches in any detail. What is important is that you can use some of the ideas presented here to help you approach the task of identifying the most important features or facilities that your proposed system will provide, taking into account the needs of all the stakeholders that you have identified.

## 9 References

Jackson, M. (2001) *Problem Frames: Analysing and Structuring Software Development Problems*. New York, ACM Press.

Leffingwell, D. (2000) 'Features, use cases, requirements, oh my!' Rational Software White Paper.

Polya, G. (1957) *How to Solve It*. 2nd edition. New York, Doubleday Anchor.