Chapter 5

Analysis Inputs

This chapter will:

- identify the resources that should be available to us as we start the analysis activity;
- introduce requirements – one of our most crucial inputs – both in the guise of traditional requirements and requirement via use cases;
- introduce patterns.

It's always a good idea, when undertaking any difficult endeavor to marshal your resources, your inputs. What do we have available to us as we begin analysis – subject matter analysis specifically? We have:

- requirements
- subject matter
- existing systems
- architectural vision
- mock-ups
- patterns

5.1 Requirements

Requirements capture, requirements analysis, requirements management and requirements tracing are all important. Sufficiently important that they deserve whole books in their own right. As the chapter title suggests, our interest as analysts and designers is getting acquainted with what we expect our requirements to bring us as a necessary input. If you

would like to go further into the subject of requirements, you could start with one of Gause & Weinberg's excellent books on requirements [e.g. Gause & Weinberg 89], or look at Tom Gilb's classic [Gilb 88].

We've argued that we can't just start coding, we need a plan – a design. And we can't just start technical planning without some input from the subject matter – analysis. Can we just get started on the analysis? No. We will have questions of relevance and we will have questions regarding functionality. A strength of the analysis activity is that it is definitive, but it is definitive only as to content not to extent. Picture an unusual first meeting between a sponsor and a developer. The sponsor hands the developer an authoritative and readable textbook on chemistry, along with the phone numbers of some expert chemists, and prepares to leave in the apparent expectation that the developers should now get on with the development of a system that they need. You would not be at all surprised that the developers are somewhat astonished, and that a couple of jaws have dropped.

They obviously have some rather important questions of the sponsor. The burning questions (after "how much are you going to pay?") will be, "What system? What is the system for?", crucial and obvious questions. Of course, in real life it's not always so obvious that a development team might start developing a system without really knowing what it's for, what is required of it, but it happens.

Sometimes there is no sponsor. When might there be no sponsor? In many of the big science systems that I've been involved with – space, astronomy and physics for example – the developers are developing solutions to their own problems. Sometimes systems are sponsored internally, by a marketing department for example, in anticipation of the needs of potential customers.

There will always be requirements however. The difficulty when there is no external sponsor will be in separating the requirements from the development; and they are always worth separating.

If you are your own sponsor, you must try to develop a split-brain approach. "Let's try to capture, in a focused, separate document that avoids analysis and design, exactly what this system must accomplish – exactly what we require of it – in advance of the development activities themselves." Put it in the guise of role-playing: picture that the situation has changed, that the software must be developed by an external company, and that you're having to tell the external company what they must achieve. You may have to push the exercise further still and imagine that you are unable to talk to the external company's programmers, that you must talk to their legal department, and thus cannot proceed by describing the algorithms you would have written.

If you are dealing with the marketing department, and trying to make sense of, or get sense from, people who want to run ideas up flag-posts to see if anyone salutes them, or who want to put a possibility in front of the cat to see if it laps it up, or who seem excessively concerned with the eventual cover design of the user manual, then you will need specialist help – in *requirements capture* or *requirements elicitation*. Try the Gause and Weinberg or Gilb books mentioned at the start of this section.

"Requirements" is a difficult term to pin down. Many documents have "requirements" in their titles. It is possible, for example, to talk about the software's requirements of the platform, which would be a concern of the developers, but which are not the requirements of concern here. The requirements that are under consideration here, as input to the analysis, concern the sponsors' and stakeholders' expectations of the accomplishments of the system-to-be – what they require of it in terms of a positive contribution to some need they have.

I like to organize any document title with the word "requirements" in it so that it clearly says a) what it is that has the requirements, and b) what it is that they require these things of. For example:

- Application Software's Requirements of the Operating System
- Application Software's Requirements of the Human–Computer Interface
- End-user Requirements of the Human–Computer Interface
- Sponsor Requirements of the Functionality of the System-To-Be

That last one is what concerns us in this book. There are clearly many other kinds of requirements important to someone, however, such things as security requirements, delivery requirements, ergonomics requirements, service level requirements, color-scheme requirements and so on.

The requirements of concern for us are sometimes known as the functional requirements. The definition of functional requirements is somewhat circular I'm afraid: they are the requirements that are essential input to the analysis and design. When non-functional requirements are wrong, the system might be too slow, or late (or early), or give you a headache if you use it too long, but it would work, it would do the job. When the functional requirements are wrong, the system doesn't work, it doesn't do the job.

Our analysis input requirements form an interface and a contract between the sponsors and stakeholders, and the developers. It is not surprising then that many – the majority even – of the problems are non-technical; they often involve social and political concerns, and are therefore something of a specialized subject. Requirements are also a key tool for project control and management. Project management can be a somewhat nebulous discipline, but two very specific things a project manager can do, that will make a big difference to success, are getting in control of requirements, and getting in control of risks.

5.1.1 Separating the requirements

Requirements capture is potentially the most definitive activity in a development. For that reason alone, it is worth ensuring that it is a distinguished phase with distinguished documentation. We don't want the speculation of the analysis (speculation as to relevance) or the speculation of the design (this invention or that invention) polluting the pristine, "facts" of the requirements. If they do, you've thrown away a chance of a really solid beginning to the project.

One upshot of this, again with a socio-political impact, is that a requirements document can sometimes be a single page with a single paragraph. Earlier we described a fingerprint project (page 48) where the requirements were very straightforward. It can be somewhat daunting to have to present a one-page report to the boss, having been charged with responsibility for the requirements. We are all prone to want to make a report look more important by making it bigger. While using heavier paper, larger margins and bigger fonts is a bit pathetic, the addition of padding text is actually dangerous to the health of the project.

The padding could be all sorts of rubbish but the biggest danger is that it will contain ill-thought out, half-baked analysis or design. We want solid, factual requirements, and we want a fully baked and separate analysis and design.

Theoretically and ideally the requirements are the sponsor's responsibility. In a perfect world, our sponsors would come to us and say, "Here is what you must accomplish with the system-to-be, here is your technical contract." And in that perfect world, the document they would plonk down on the table would be complete, implementation independent, unambiguous, understandable … .

Well, we must open our eyes now, and ask about the real world. We developers are often going to have to wade in and capture the requirements ourselves, or help the sponsor to capture the requirements, or help the sponsors document the requirements in a usable form.

You've probably figured it out by now, but why do I insist on saying "capture", in *requirements capture*? It's just to emphasize that they are out there. We developers don't *invent* them, we don't *model* them. We capture them.

Can we have such a thing as object-oriented requirements? You may have seen such phrases; I have. While the requirements will concern "objects", it will only be in the most trivial sense. In no sense will we want, or gain any benefit from orienting the requirements around objects; around instances, around classes, around messages, around polymorphism. The requirements are the requirements. There is no point in re-orienting them.

5.1.2 The nature of requirements

The requirements answer the question, "What is the system for?" They tell us why we are bothering. They tell us why the world will be a better place with the installation of the system-to-be. They tell us what we must accomplish with the system-to-be else consider that we have failed. They give our principal input for answering questions of relevance during the analysis and design. During design, they lead to the answers to the difficult questions as the functionality of the system-to-be.

Requirements are a sponsor artifact and requirements should be kept separate. Putting those together gives us a very good guideline for requirements: requirements concern the system boundary. If your requirements are not talking about what goes in and what comes out, then they're not exclusively requirements.

There is another very good guideline that we have mentioned elsewhere. The requirements of interest here are the only part of the development that are truly implementation independent. Check that the requirements documents would be *absolutely unchanged* if the anticipated technology were to suddenly and radically change. "Oh, by the way, we won't be doing an OO system in Java, we've decided to use a neural network." Or, "We've decided to give palm-tops to all the field personnel." The only implementation dependence is that requirements we can meet have to be technically feasible.

Classic requirements

Some of the best and most useful requirements I ever encountered were those that consisted of a set of atomic statements about what the system must do, should do or might do.

Say we were presented with the following requirement: "The system must take a facial image, in any of the common image formats such as, today, JPEG, …, TIFF, produce a numerically-based profile and find other pictures, from selected databases, that are likely to be of the same person."

The idea of the "must …", "should …" and "might …" is to prioritize. It is important to the focus of a project that the most important requirements are at the top of any list. The details of the classification scheme can vary, but it's fairly obvious stuff.[1]

The idea of the "atomic" is to avoid combinatorics. Once you start trying to describe scenarios that combine different requirements, you hit the combinatoric, or factorial explosion.[2] So we break any combinations up, looking for the irreducible set of requirements. Such a list will be completable and of a manageable, though variable, size.

Our example of three paragraphs ago could be improved. It actually combines three requirements, which should be separated:

- "The system must register the location of, or record the content of, images in any of the common image formats; today including such formats as JPEG, …, TIFF."
- "The system must take a selected facial image, and express the characteristics of the image as a numerically-based profile that will serve to identify that face among all possible faces."
- "The system must take a facial image numeric profile and find all likely images of the same face from selected image sources."

Notice that, now, those three sentences (and sentences is what we are aiming at) are irreducible or "atomic".

1. Just don't try to do an absolute ordering. Three or four categories are all that most projects can manage.
2. Factorial growth is fast, really fast. How big is n, if n factorial (that is $n \times n-1 \times n-2 \times \ldots \times 1$) often written as $n!$, works out to the number of seconds since planet Earth was formed? Would it be 1000!, 500!, 100! No, just 20!

Some people worry that classic requirements are expressed in words rather than in diagrams. While a picture is frequently worth a thousand words, it isn't always necessarily so. The requirements could involve *absolutely anything.* To try to find, and impose, a pictorial form is doomed to distortion and failure. It will be when we start modeling that we start to find pictures indispensable.

We must, therefore, be on our guard against the main drawback of text – its imprecision. Choosing atomic requirements helps. What else can we do? We can put effort into the choice of words. We will have mentioned elsewhere that any writing or meeting activity that doesn't have a dictionary and thesaurus within arm's reach loses several marks. We can make sure that at least one of the people involved in the writing, and perhaps reviewing, has a passing knowledge of grammar and clear expression (without being a tiresome pedant).

Notice, by the way, how the phrasing improved in the re-writing of the requirement exampled a moment ago.

We can also quantify the requirements. This is one of the ten most important things you can do to improve a project's chance of success. Look into Tom Gilb's writings [Gilb 88 for example].

- "The system must register the location of indicated images, using something at least as concise and precise as a URL, or record the content of images in any of the common image formats, including today such formats as JPEG, …, TIFF, to any resolution tending to be encountered, which today would be varying from 75 dpi to 1200 dpi. Formats need not be restricted to raster formats; vector formats may also be employed."
- "The system must take a selected facial image, and express the characteristics of the image as a numerically-based profile that will serve to identify that face among all possible faces. A false positive rate of x and a false negative rate of y are the maximum acceptable errors. The numeric representation of the image should take no more than twice as much storage as the original image required (negotiable)."
- "The system must take a facial image numeric profile and find 99.9% of all likely (see above requirement for error rates) images of the same face from selected image sources within seconds rather than minutes, certainly less than an hour on even the most heavily-loaded system."

Quantifying is demanding, time-consuming work. The above quantification isn't perfect yet. To be realistic, some projects will have to content themselves with quantifying the high-priority requirements – the "musts". One of the excellent things about Gilb's work, is that he shows us how requirements we would have sworn defied quantification can always be quantified.

Use case scenarios

The drawback to classic requirements, as described above, is that they are not very inspirational. That is to say no-one reading such a dry list gets a good mental picture of what the system is for, they don't animate the system for the human mind.

The solution is to augment the definitive, completable "classic" requirements by bringing back the scenarios we argued against only a moment ago, but to recognize that the scenarios can only possibly be a sampling, and in that way avoid the combinatoric explosion. The essential trick is to choose those scenarios that are the most illustrative, the most important, scenarios.

A *use case scenario* describes an example of the system-to-be in action. It is big enough that one forms a useful picture of the purpose of the system. It is small enough such that in the implementation, one suspects that it could be enacted by the instances of half-a-dozen object types. That last sentence needs several caveats. Don't get confused between the requirements and the implementation – between use case scenarios and something called an interaction diagram which is coming up later on, in design. Use case scenarios are part of requirements, they make sense to sponsors and users and they are an external perception. Interaction diagrams are part of the design, they don't necessarily make sense to sponsors and users and they are an internal perception. So why have I risked confusing the two? Because the requirements' use case scenarios form an excellent schedule for the design's CRC sessions that result in the interaction diagrams, provided that they have a reasonable granularity. The second caveat is that, of course, it takes experience and luck to be able to judge the "size" of a use case scenario in these early, requirements days.

We've already given an example. It was what our three "classic" or "static" requirements were derived from in the previous section: "The system must take a facial image, in any of the common image formats, such as, today, JPEG, …, TIFF, produce a numerically-based profile and find other pictures, from selected databases, that are likely to be of the same person." Sometimes it comes about that one thinks of use cases first and then derives the irreducible requirements from them; and sometimes one thinks of the irreducible requirements first and then animates them via scenarios. Either order is fine.

A use case scenarios is usually dressed up a little bit:

- We typically highlight the event that starts it off, although we don't have to. The main reason for doing this is to ensure that the scenario really is helpful later in the CRC sessions (see *Getting started* on page 272).
- We usually give titles to the use cases, and often reference numbers (although if you don't have a tool helping you with them, reference numbers usually end up being more of a hindrance than a help).
- We can also highlight the target output, or the target state change, of the scenario. Any meaningful and useful description of the system in action will result in output, state change or both. Although we want the use case scenario to concern itself only with that which is apparent from outside the system-to-be, if information has been consumed

and not necessarily all been output again, the system must have retained something, must have changed state. Once again, part of the reason for highlighting this, is its future usefulness in the CRC sessions. We are careful to note the state change in abstract terms rather than in any anticipated updates to the variables of the system-to-be.

Figure 5.1
Example use case scenario

Use case name:	Locate facial images matching a poor target facial image	Use case reference no.:	
Super use case:			
Initiating event:	Target image selected		
Key outputs:	Set of matching facial images and their locations		
Key state changes:	None		

Script:
The system offers a filterable and searchable list of available image files. We select one image as the target image that we want to match, and we select a set of image files against which the target is to be matched and our required degree of confidence. The system displays the image attributes, the details of which are not important to this use case. We confirm that it is the image we desired. The system warns us that it considers that there is a problem with the target image file – in this example, that it considers the image not to be a facial image. We elect to go ahead anyway. The system indicates its best guess as to the target image's salient features – in this example, the nose and one ear. The system finds three possible matches and displays them.

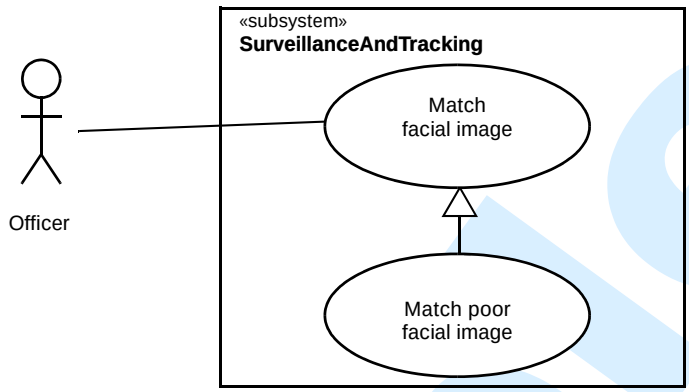
The most important part is the scenario itself – the script or storyboard. How will that be depicted? A textual script is often entirely adequate, once again being careful to express it in terms of what is apparent from outside the system-to-be. If we are doing lists of irreducible requirements as well as use case scenarios, then we should almost certainly have a scenario reference the irreducible requirements that it covers; and we would typically put the bulk of our effort over clarity and precision with words into those irreducible requirements. Figure 5.1 shows an example use case.

Sometimes text will not be adequate for a scenario's script. First of all ask yourself if that's because the use case is too big. If that's not the reason for the complexity, then consider an activity diagram. We have mentioned these "grown up" flowcharts already (*Business systems analysis* on page 14) and will be detailing them later (*Activity diagrams* on page 218). Whatever you chose, remember that there is no pressing reason for requirements descriptions to have an object orientation.

You might have heard that the UML includes use case diagrams, and might be wondering why we haven't mentioned them for the depiction of the script.

The UML's use case diagrams, for example Figure 5.2, describe the organization and context of the use cases; they don't describe the use cases themselves.

Figure 5.2
UML use case diagram



In the example use case scenario of Figure 5.1, you may have wondered about the entry for *super use case*. This covers a situation where one discovers that several use cases have elements in common, and that the commonality could fairly be described as an *is-a-kind-of* commonality. As with many other models and UML diagrams one can draw out that commonality into a super something, and depict the relationships between sub and super use cases with arrows. There are other relationships between use cases in UML use case diagrams; they are described in the UML appendix (Appendix C, on page 467).

Use cases are very useful, as we've said. Use case *diagrams* can be useful for organizing large numbers of use cases, but in and of themselves, in my opinion at least, they don't contribute much to one's understanding. One could therefore say that the functional requirements can often get by quite happily with use cases but without use case diagrams.

How many scenarios would we describe for our requirements? I honestly don't know at present. I'm afraid my answer boils down to "until you run out of the allotted time, so make sure you really do start with the best scenarios". I am only sure that less than half-a-dozen scenarios would be inadequate for most systems, and that more than a couple of hundred scenarios would be excessive for most systems.[3] Part of the problem is that systems are hugely variable. As we've mentioned, there can be systems with a single use case at one end of a spectrum; in a transaction processing system there could be large numbers.

5.1.3 Staged development

Back when computers were novel, and computer systems were smaller and less ambitious, the typical development and delivery style was what we now term "big bang". The developers developed the system and they delivered the whole thing on, or near, or all too frequently way after, the appointed day.

3. And if you are contemplating hundreds of use cases, you might do well to read the warnings collected by Steve Palmer. Steve sounds as though he seems to be against use cases, which I am not; but many of the warnings are sensible ones [Palmer 02].

Today, the chances of being able, in a single delivery, at the very end of the development period, to spring a new multi-million line of code system on the hapless customer, after having spent a couple of years out of sight and mind are low to vanishing. Instead, we prefer to develop and deliver today's typical projects in stages – *staged development and delivery.* We mentioned this when discussing packaging earlier on.

Staged development is not to be confused with prototyping. A prototype in this book has the same meaning as it has always had for engineering. It's the wooden car that was built to check the airflow patterns and that you wouldn't even consider trying to deliver to the customer; you throw it away or you put it in a museum. I'll just repeat that. *A prototype is thrown away.* There is a section on the perils of prototypes toward the end of this chapter (see *Mock-ups* on page 79).

Staged development involves no short cuts in material or quality. The components of any stage are regular, quality components. We simply have a desire to deliver anything that can be delivered, as soon as possible, because then it can be evaluated and validated as soon as possible. And if the development has taken a wrong turning, we will find out as soon as possible, for as little wastefully spent money as possible.

Staged developments require organization and care. The organization of staged development can begin here in the requirements. There are many ways to organize the requirements: importance was one way; staged development – delivery time – is another. It is important to note that importance and delivery stage are different: one might need to deliver the most important requirement as part of the last stage.

Incidentally, the requirements' many classification possibilities mean that the way in which the requirements are presented is very important. A tool is almost essential, and it should be capable of maintaining several different navigation threads through the requirements, or at least being able to sort and resort on several distinctly maintained criteria.

5.1.4 Change cases

It can be worth including some requirements that are not planned for any of the development stages, known as *speculative requirements* or *change cases.* They give us input when we are contrasting design alternatives. They give us tests of the malleability of the emerging design. Every so often, two or three experienced designers sit down (every second Friday for a couple of hours, for example) and consider the impact on the emerging design should a sample of these requirements actually have to be implemented. Would the design nod sagely and need just one or two changes and additions? Or would the design fall to pieces?

5.1.5 Checking and re-checking the requirements

Requirements are not usually easy to get. And requirements are not easy to get right. Putting it another way: requirements will typically be incomplete and have mistakes.

Requirements evolve and it's no good us pretending they don't. Sponsors will change their mind, or the sponsors' subject matter might change their minds for them. Although there are control issues, contract issues and price issues that are outside the scope of this book, developers must assure themselves, from time to time, that the requirements they are working to, are still the correct requirements.

If the requirements don't seem to be sensible, it's a bloody-minded developer who would decide to follow them come hell or high water (or a bloody-minded management who would insist that the requirements were to be treated as immutable tablets of stone).

5.1.6 Checking the requirements coverage

(You could skip this section if you're in pursuit of the basics.)

There's something else that would be really great to do. That would be to check that the system being developed is actually doing what was required of it – checking requirements coverage as it is usually termed. Even better would be to formally trace the requirements' manifestations into and through the various models of the development, including the final, executable model (the code).

You'll have sensed a "but" coming up, no doubt. Here it is. Formal requirements coverage analysis and tracing are really, really difficult. Of course we informally check that we are covering the requirements, in that they guide and feed all of our development models. But formal analysis of their coverage and formal tracing to their realization is beyond level 1 and level 2 organizations (and is very challenging for level 3s). (See page 9 for a quick sketch of these capability maturity model levels.)

So why have I mentioned it in this book, where requirements are not a primary issue, and where we want to keep a practical focus? I've mentioned it because level 1 and level 2 organizations do sometimes *try* to do, or are told to do, requirements coverage and tracing but it takes up vast amounts of their time, there is a false sense of comfort, very little is achieved though and a large amount of time and effort will have been spent to no avail.

5.1.7 Recognizing the limits of requirements

Finally, as far as the requirements are concerned, a reminder of something that was mentioned in the three-model proposal. You will encounter the phrase *requirements analysis*. If you do encounter it, ask yourself exactly what is meant by it. Both the term "requirements" and the term "analysis" are used in diverse ways, so it could mean anything. If their meanings are similar to their meanings in this book, then be very careful with any implication that requirements analysis is the one and only phase that precedes design. The position taken in this book is that requirements are just one of the inputs to a subsequent, analysis phase.

If "requirements analysis" is simply the term for ensuring the completeness and quality of presentation of the requirements, and if there will be another phase between requirements and design, no problem.

Let's look at the other inputs to the next phase, the analysis phase.

5.2 The Subject Matter

You will already know the main input to the analysis, from the three-model proposal of Chapter 2. Our main input is the subject matter. It's always there. We should always study it. For historical, or even sentimental reasons we have decided to term this study "analysis" – *subject matter analysis.* It might present itself to us via users, sponsors, experts, standards, reference works or investigation. It is the subject of the next two chapters.

5.3 A Previous or Existing System

Slightly less has been said already about another possible input: *systems analysis.* In this book we have been careful to distinguish requirements from analysis and design, and we are being careful to distinguish subject matter analysis from systems analysis. Where there is an existing system, i.e. an existing systematic solution, in addition to the subject matter, and where the existing system is amenable to analysis – it's not an existing computer system for example – and where the analysis of the existing is reckoned likely to provide useful input to the design, then we can do systems analysis.

Systems analysis is well understood. This book doesn't have a lot of novel input to systems analysis. Systems analysis need not always be object-oriented, sometimes it cannot be object-oriented, frequently it can be object-oriented, and usually if it can be object-oriented it should be object-oriented.

Systems analysis is the subject of a later chapter.

5.4 An Architectural Vision

The term "architecture" is probably over-used. It can be used almost as a synonym for the structures of the design – the software architecture. Here I mean the system architecture. Is this a system that will run on a standalone Mac? Is this a four-tier system with EJB forming the major framework for two of the layers?

Sometimes projects are exploratory. One might not be certain that a project can be done at all. One might not have decided how best to deploy the software on a given platform,[4] or even what the candidate platforms are. Usually however, one of the characteristics of successful projects, especially large successful projects, is that there is an architect with a vision. In a large project it might be a (small) architecture team. Someone, somewhere, must have, or be forming, a picture of how it's all going to hang together. If that isn't the case, then their absence must go high up on the risk schedule.[5]

4. A platform is taken to be a combination of computing machinery – IBM, Cray, Intel PC, Apple … – operating system – z/OS , Windows, Linux … – and possibly a framework such as Enterprise JavaBeans.
5. An essential part of the control of a project is to know, document and manage the risks.

If your project is experimental, architectural proposals and decisions are key milestones.

As one of the developers at whom this book is aimed, your job is to find the architectural proposal and be guided by it in part. During the analysis, the proposed architecture will provide input, for example, to decisions concerning relevance and importance. During the design, the proposed system architecture will help, for example, in choosing between alternative software micro-architectures.[6]

5.5 Mock-ups

We have just been talking about architectural vision. A mock-up can be a valuable part of establishing or validating an architecture. With a human-used, graphically-interfaced system, it's relatively easy to create a mock-up to give people a good feel for the proposal. A technical mock-up can establish whether critical design choices are likely to work.

Mock-ups, especially when being shown to users and domain experts, can provoke fresh insights into the subject matter.

It was hinted earlier, that there were potential problems with prototypes, however, and that's partly why I'm calling them mock-ups rather than prototypes.

The problem is that, in software engineering, prototype has all but lost its original meaning of a mock-up that uses alternative materials in order to quickly or cheaply answer questions. Prototypes have become confused with delivery stages. Imagine a project manager seeing a foamboard mock-up of a building in their architects' foyer, their eyes lighting up and they're saying, "Great, you've nearly finished. Let's slap a coat of paint on it and then we can at least deliver something to the customers, who are baying outside my door." That happens all the time in software projects.

If you are not going to throw away your prototype, you will certainly find yourself wanting to throw away version 1 of your "product", when the time comes to contemplate its evolution into version 2. So:

- Call them mock-ups. Never use the word prototype.
- Never let mock-ups be seen by anyone except the beneficiaries, like end-users or test engineers; certainly don't let them be seen by sponsors and managers.
- Don't fall into the trap of taking pride in how good your mock-up looks – so good it could be taken for the product. Make mock-ups that are clearly broken outside of the area of concern. Have them crash frequently and be agonizingly slow outside of the area of concern. Have screens that use vile color schemes and clunky widgets, if your mock-up isn't being used to validate interfaces.

6. However, the job title "architect" can cover a multitude of sins, so let's be clear that the kind of architect I have in mind is more technically savvy rather than less, has a real grasp of the totality of a system rather than being out of touch, is a mentor and a coach rather than a recluse, is a visionary and a leader, and is rightly held in respect if not awe (and probably in golden handcuffs). While envisioning imagineers do have a role to play, they aren't necessarily architects.

5.6 Analysis Patterns

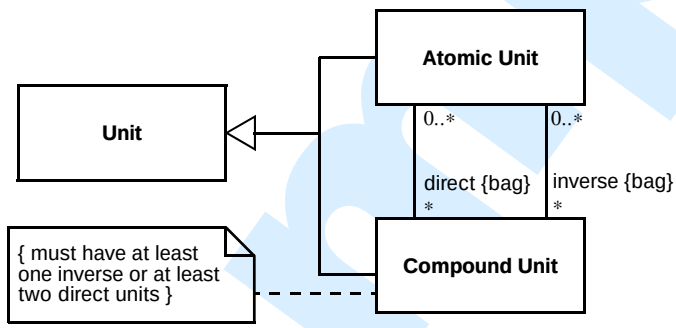
Our last input is formed from things we have learned from previous analyses.

The difficulties of reusing an analysis model have already been touched on elsewhere (page 65). There were similar difficulties when we tried to come up with standard system architectures. The attempt to write the architecture handbook gave rise to the very important design patterns (Section 10.5 on page 259). And if we can't take a successful analysis model down from the shelf, we can at least be aware of the existence of analysis patterns.

Just as design patterns are reusable technical micro-models, analysis patterns give us reusable micro-models to use in our analysis models. Martin Fowler wrote the seminal book on analysis patterns [Fowler 97]. Make sure this book is in the project library.

For example, if units and measurements are centrally important to your model, how do you represent them? Thousands of analysts have struggled to put units into models. Breathe a sigh of relief, they are in the catalog; Martin Fowler suggests that compound units can be modeled along the lines of Figure 5.3, for example:

Figure 5.3
An analysis pattern for units



I have kept the original {bag} constraint. Currently UML 2 is not totally clear about representing the different kinds of collections. A bag is a set that relaxes the uniqueness constraint and allows duplicates.

5.7 Case Study

5.7.1 Requirements practice

As already mentioned, this book isn't trying to cover the subject of requirements fully, but if you would like a little practice in creating or assessing requirements, you might like to try writing down a few "atomic" requirements of the restaurant system – "The system must … The system shall … You might like to write down two or three use cases, giving the initiating event, the script (storyboard) and any key outputs or state changes (you might like to remind yourself about use case descriptions by looking at Figure 5.1, on page 74).

You might also want to look back at the introduction to ALICE on page 19.

! Turn the page when you have your answer

I hope you didn't start with something too vague. It would be no good writing down:

- bookings
- ordering
- etc.

If anything, those would be section headings in the document. And it would be no better if you had added a forbidden word like "handle":

- handle bookings

"Handle" is forbidden because it's "noise"; it adds no meaning. We will come back to the forbidden words when we get to the main subject matter analysis chapter.

A typical first thought goes something like this:

- The system must take booking details and find tables.

That's better but it's too big to be an "atomic" requirement; and it's too small to be a use case scenario. Can we break it down while retaining meaning? Yes:

- The system must take/input/register/record the details of a requested booking.
- The system must locate a table of suitable capacity for a given booking (or a table that is minimally bigger than the party size), a table that has the required characteristics – for example, but not limited to, smoking/non-smoking and quiet, and a table that is free at the time in question.
- The system must allocate a given booking to a given table, advising of any unsuitable or impossible characteristics it discerns such as, but not limited to, a table that is too small, too big, not free, or not non-smoking.

Note the search for the best word in the first bullet point above. "Take" and "input" are too vague; "register" better connotes what we want.

You might wonder whether the requirements need to define what a booking is. The danger would be straying into the territory of premature analysis. I tend to take the same attitude as a good legal contract would take. There are certain words that "the common person"[7] would understand and that don't require the contract to define them; and there are certain important words that do require definition. For those words that do require definition, I avoid straying into analysis by starting their definition in a glossary and leaving an open cross-reference that will eventually lead to the most appropriate definition of a booking in the analysis documentation to come.

Note the use of the phrase "for example, but not limited to". Some developer–sponsor relationships would find such wording suitable; others might feel that every single item must be spelled out. The danger with trying to spell everything out is that something will inevitably be missed and unless there is a fairly continuous system for updating require-

7. An alternative legal phrase, believe it or not, is the "man on the Clapham omnibus", a phrase I particularly like as I lived in Clapham.

ments – which is too onerous for many projects – the developers can quite rightly say, "Sorry – it wasn't in the requirements." The requirement in this case could be cross-referenced to another requirement:

- The system shall understand, and use in matching bookings to tables, a set of booking characteristics that is user-configurable, and could include such things as smoking, non-smoking, quiet and good legroom.

The three booking requirements that got broken out above now give us the constituents of several possible use case scenarios:

- A telephone call is received requesting a booking for a certain time on a certain date and for a certain number of people. A non-smoking table is requested. The details of the requested booking are input to the system. The system suggests a suitable table which is accepted by the user whereupon the booking is allocated to the table.
- The user inputs the details of a booking to the system. The user allocates said booking to a table. The system warns that the table is bigger than the booking requires. The user effects the allocation despite the warning.
- The user inputs the details of a booking to the system. The user allocates the booking in question to a table. The system indicates that the table is not free at the time of the booking and refuses to make the allocation. The user examines that table's bookings before and after the booking in question and modifies their times. The booking in question is then allocated to the table.

Note that a scenario should be a particular, typical, important and useful example.

The interaction between atomic requirements and use cases can go both ways. The last two use cases bring to light other atomic requirement:

- The system must be able to locate significant entities such as bookings, orders, tables, members of staff via their properties or their relationships.
- The system shall, when indicating to the user such things as tables, bookings and orders, use an intuitive and appropriate metaphor such as screen forms for the bookings and orders and two-dimensional approximated physical position for tables.

Notice that every requirement has described only what the sponsors and users can see, exclusively an external perception. Later on as a design exercise we will plan and illustrate some object interactions. They will illustrate internal perceptions. Some of them will match use case scenarios illustrating how the objects-to-be will enact requirements.

Notice that little has been mandated as far as the technology is concerned. Our requirements are implementation independent.

Figure 5.4 lists the set of requirements that will guide the rest of the case study. Each of the critical requirements could be preceded by the phrase "The system must …"; each of the great-to-have requirements could be preceded by the phrase "The system should…"; and so on.

Figure 5.4
Case study irreducible ("atomic") requirements

Critical (must)

Register the details of a booking request.

Allocate a given booking to a given table, advising of any unsuitable or impossible characteristics it discerns such as, but not limited to, a table that is too small, too big, not free, or not non-smoking.

Locate a table of suitable capacity for a given booking (or a table that is minimally bigger than the party size), where the table has the required characteristics – for example, but not limited to, smoking/non-smoking or quiet – and where the table is free at the time in question.

Note updates regarding table availability to reflect such things as "walk-ins" (diners who have not booked).

Register changes to booking details advising of any consequential unsuitability or impossibility.

Flexibly present the details (at least including the contact name) of bookings expected within a selected time period, including now, for one or more selected tables.

Register the arrival of a party who have booked thus updating the table to an occupied condition.

Register the details of a meal order, including but not limited to department (course), quantity of servings, preferences (including for example rare/medium/well-done or no dressing).

Determine the allocation to kitchen departments (starters, mains, etc.) of meal order contents.

Notify departments of newly taken orders via the output the system is configured for (e.g. printout).

Register changes to the details of a meal order, alerting when items being changed appear to already have been committed to kitchen department or served.

Produce bills.

Register the details of a dish, including name, description and price; and any preferences that must be determined by the order taker (e.g. waiter/waitress) such as rare/medium/well-done.

Register changes to the details of dishes, drinks, etc. including the name, description, price and the likely number of servings of a dish or bottles of a wine that remain to be served.

Be able to locate significant entities such as bookings, orders, tables, menus, dishes, members of staff via their properties or their relationships and portray their details in summary or in detail.

Register updates to any standing information such as normal opening hours, nominal table capacities, table numbers (IDs) and table characteristics such as extra legroom.

In general, register changes to the details of any significant entities such as bookings, orders, tables, menus, dishes, members of staff.

Assemble (via user choices) a new menu from old menus and/or selected dishes, supporting the allocation of dishes to their appropriate place in the menu (typically starter, main or after) and their kitchen department (often but not always the same – starter, main or after).

Prepare selected menus for output (e.g. printing, rich text format or dynamic HTML), including the setting up title, section headings and legals (e.g. service not/is included); there should at least be the options of: same as used yesterday, choose from recently used, edit chosen or new.

Register the details of a bill payment, including date and under- or over-payment (tip).

Flexibly recall and present current and past menu details.

Flexibly recall and present recipe details.

Flexibly recall and present floating, placed and cancelled bookings' details.

The DSDM approach[8] popularized the acronym MoSCoW: *must*, *should*, *could* and *won't*. The "must" requirements are non-negotiable, they form a coherent set and they cannot be "cherry-picked". The "should"s are those requirements that would be great to have if at all

8. DSDM (www.dsdm.org) is a development approach based on pragmatism, which is laudable. It is pretty much centered around rapid application development, which is more questionable. DSDM itself says that it is based on common sense. I'm not sure what that is. (Einstein said that "Common sense is the collection of prejudices acquired by age eighteen.")

Figure 5.4
(continued)

Great to have (should)

Understand (and use in matching bookings to tables) a set of booking characteristics that is user-configurable, and could include such things as smoking, non-smoking, quiet and good legroom.

When indicating and presenting to the user such things as tables, bookings and orders, use an intuitive and appropriate metaphor such as screen forms for bookings and orders, calendars for bookings, and two-dimensional approximated physical position for tables.

Register, update and recall the allocation of waiters to tables and changes thereto.

Report time to deliver a dish either by using recipe information or by using any kitchen or table preparation times registered for that dish.

Raise an alert when the likely number of servings remaining has been recorded for a dish, and a meal order requests servings in excess of that.

Nice to have (could)

Calculate approximate stock depletion from meal orders placed.

Change case (won't)

Notify stock items below minimum levels on request.

Notify stock items near expiry.

Register stock level corrections.

Register stock item information such as quantity-on-hand, normal unit, maximum level, minimum level, re-order level, typical re-order quantity.

Register stock acquisitions.

Accept the booking of a party to more than one table.

Support the analysis and correlation of takings, including but not limited to takings by table, time, date, season and menu.

Accept bookings for the entire restaurant.

Accept bookings for an entire floor of the restaurant.

possible. The "could"s are the requirements that it would be nice to meet as long as nothing else is affected. Finally, the "won't" requirements are those requirements that won't be implemented this time, but which might be implemented in a future version. The "won't"s are the useful source of change cases.

One of the "atomic" requirements in Figure 5.4 is rather vaguely worded and its tightening up has been included as an exercise at the end of the chapter.

Figure 5.5 has three example use case scenarios.

Exercises

5.1 How many different kinds of requirements can you think of? List them. For example, you might include functional requirements and security requirements.

5.2 Classify the following requirements using your answer to the previous question. You can improve your answer to the previous question if necessary. (I am sure that the wording of these requirements could be improved as well.)

- "The system must never be unavailable for more than three minutes."
- "The machine room must have less than 0.1 ppm ozone."

Figure 5.5
Use case scenarios

Use case name:	Successfully make a system-assisted booking	Use case reference no.:	
Initiating event:	Customer contacts restaurant, requesting a booking		
Key outputs:	None		
Key state changes:	New booking added to system		

Script: A booking request is received (e.g. by phone or over the Net). A day, a start time and a party size are given. Preferences may be given also. In this scenario, it is a non-smoking, quiet table that is requested. A duration could be given, but in this scenario, as is common, no duration is given.

In this scenario, the customer is not recognized; there is no favorite table lookup.

The details of the requested booking will be input to the system.

The system will locate a suitable table. That means a table that is the right capacity for the party, or a table that is minimally bigger than the party size. It means a table that has the required characteristics – non-smoking and quiet. It means a table that is free at the time in question.

The system will display the suggested table; a table *is* found in this scenario. (The GUI or the dynamic web page might display the location of the table, but that is an included sub-use case, described separately.)

The user confirms that the table is OK. In this scenario, the table is OK.

The system allocates the booking in question to the table in question.

Use case name:	Successfully locate a booked table on party arrival	Use case reference no.:	
Initiating event:	Part arrives claiming a booking was made		
Key outputs:	None		
Key state changes:	1) Table noted as being occupied 2) Booking noted as honored		

Script: The name in which the booking was made is obtained from the party. The system displays the contact names of the bookings due to arrive within 20 minutes either side of the current time. The party's contact name is found. The system indicates which table was allocated. The system is informed that said table is now occupied and that the booking was honored.

- "The system must calculate the optimum angle of attack for the aft hydrofoil."
- "The system must be operating to the downtime requirements by 0001 hours 11 December 2003."
- "The system must pass a FAST (Federation Against Software Theft) audit – no unlicensed software for example – and must not require registration under the Data Protection Act."
- "The system must play back at least 48 tracks of 24 bit/96 KHz uncompressed audio data, with one digital effect (e.g. reverb) per channel, simultaneously."

Figure 5.5 (continued)

Use case name:	Meal order using hand-held terminal	Use case reference no.:	
Initiating event:	Table (party) ready to order		
Key outputs:	Dish lists per department		
Key state changes:	1) New order added to system 2) Table noted as awaiting starters		
Script: The first diner gives his starter and main. The system is informed. The second and third diners each give their starters. The system is informed. The first diner changes his mind and requests the same starter as the second diner, but without the parmesan flakes. The second diner gives her main course. It is steak and the system prompts the user to request the preference as to rare, medium or well done. The third diner gives their choice of main. The user reads back the order and the diners confirm. The user confirms to the system. The system registers the order and the time taken. The different dishes are split out according to the department that prepares them. The different dish lists are output (e.g. printed) at the different departments, annotated with the time taken, the table and waiter/waitress.			

5.3 Break the following scenario down into "atomic", i.e. irreducible requirements.

"We would typically call up yesterday's menu. We would check with Chef to see if any particular stock items need using up via any particular dishes. I guess that maybe the system could help us with alerts as to any expensive stock approaching its use-by date. We would amend the menu to remove dishes not on offer today and to add the new dishes like the stock-using dishes and any other dishes that the manager or Chef have planned."

(You) "Would you type up the details of each new dish?"

"Well I guess there might be totally new dishes; but mostly we would want to call up some kind of overall dish list, where the typical price, description, etc. was already entered. Sometimes we would have to amend the defaults – for expensive ingredients whose prices fluctuate wildly, for example."

(You) "And would you be changing yesterday's menu to become today's? Or would you be leaving yesterday's menu on record and creating today's from it?"

"Oh, we would definitely want to keep a record of previously used menus; we'd simply be using yesterday's as a typical starting point. I imagine there would be times, though, when we'd create a menu completely from scratch."

5.4 Given the above dialog, what extra questions would you have asked if you had actually been there?

5.5 Rewrite the following requirements of POIROT and ALICE to be clearer and more precise. They don't necessarily have to be correct. In other words, your job is to make them less ambiguous, you won't necessarily have the information to make them more correct.

- "The system must accept crimes."
- "Match fingerprints." (Hint: the term "match" is subjective. Some countries require as few as 8 points of similarity while the UK requires 16.)
- "Produce bills."

5.6 (Difficult) Quantify the following requirements. Come up with numbers that the system-to-be can be measured against in order to establish if it has met the requirement.

- "The data shall be more consistent."
- "The system must register the details of a booking request."
- "The system must portray in a readily-assimilated fashion, the selected statuses (for example, location and/or readiness and/or …) of a flexibly selected sets of active personnel (for example, an individual, all personnel matching an input declarative query (e.g. SQL), or all active personnel)."
- "Usability is our number one priority."

5.7 You have limited time (as ever). Using your own knowledge of staying at hotels (or using your imagination), write up two important and illustrative use case scenarios for POLLY, the hotel system development mentioned in Exercise 3.3.

5.8 What are the subject matters for the following computer packages: a word processor, desktop publishing, a sales ledger and vehicle number place recognition?

5.9 Think about and write up a SWOT (strengths, weaknesses, opportunities and threats – essentially an enhanced version of pros and cons) analysis of prototyping as a technique in software engineering.

5.10 If you have been involved in producing analysis models (subject matter models, conceptual models, real-world models, …), ask yourself if you've found yourself needing to model the same situation more than once. Did you find or invent a reusable analysis pattern? If you invented one, is there a known version (Martin Fowler's web site for example); and if there is how does it compare with your version?