FEABHAS

The Baker’s dozen of Use Cases

Glennan Carnie

The baker’s dozen of use cases

Use cases have become a core part of the requirements analyst’s arsenal. Used well they can bring dramatic increases in customer satisfaction and a whole host of other subtle benefits to software development.

The use case itself is very simple in concept: describe the functionality of the system in terms of interactions between the system and its external interactors. The focus of the use case is system usage, from an external perspective.

Despite this apparent simplicity, requirements analysts frequently struggle to write coherent, consistent use cases that can be used to facilitate development. Often, the use case analysis becomes an exercise in confusion, incomprehension and the dreaded ‘analysis paralysis’.

This paper aims to aid use case writers by presenting a set of rules to follow when performing use case analysis. The rules are designed to avoid common pitfalls in the analysis process and lead to a much more coherent set of requirements.

These 13 guidelines are by no means exhaustive (for example, what about ‘Abstract’ actors; or System Integrity use cases?) and I’m sure there are dozens more I could add to the list (by all means, let me know your golden rules) The aim of writing this article was to give beginners to use case modelling a simple framework for creating useful, effective use cases – something I think is sorely missing.

The Baker’s Dozen can be (neatly) summed up by the following principles:

- Understand the difference between analysis and design.
- Understand the value of a model – know why to create the model, not just how.
- Understand the difference between precision and detail.
- Keep it simple; but never simplistic

A brief overview of use cases

A use case is partial definition of a system's functionality, described in terms of a *goal* that the user of the system (called an Actor) wants to achieve. The logic behind this is an Actor doesn't want to use a software system just for the hell of it – the Actor wants to *achieve* something. This achievement is the Actor's goal. The system's functionality can be described by the complete set of use cases for the system.

A use case is described in terms of the interaction between the Actor and the system; specifically, it is defined in terms of the *information* that is exchanged between the actor and the system.

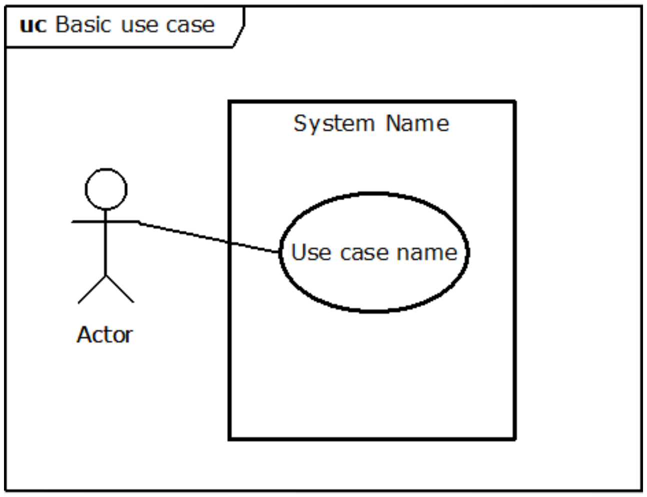


Figure 1 - The basic use case diagram

It is useful to think of a use case in terms of *scenarios*. A scenario is a particular instance of interactions. A scenario describes one possible way of interacting with the system (whilst trying to achieve that particular goal). When trying to achieve their goal, the Actor may have to make choices. Also, things have a habit of going wrong. Each variation is a different scenario. The use case, therefore, is described by every possible scenario. Obviously, attempting to document a use case by writing every scenario would be ludicrous for anything but the most trivial system. The way round this is via a *use case description*.

A use case description is a structured English definition of all the use case steps. It can be thought of as a blueprint for generating use case scenarios. It should be possible to re-create any scenario from the use case specification.

In its basic form, a use case description is a series of steps showing how information is exchanged between the Actor(s) and the system until the Actor's goal is met. This flow is always initiated by an Actor (we don't really want software doing stuff of its own accord!) This basic form is called the Basic flow. In the Basic flow nothing goes wrong and the Actor does the most normal, obvious things (this is sometimes called the "Sunny Day" scenario!)

However, when the Actor must make a choice, the use case description must fork – one path for each option the Actor can make. Each of these descriptions is described in an Optional flow (sometimes called a Sub flow).

Similarly, not everything always goes right. If there is a chance for something to go wrong, chances are it will at some point. Typically, these conditions are not resolved by giving the Actor an option; they happen beyond the Actor's control. We have to record what happens when that 'something' goes wrong. These descriptions are called the

Exceptional flows.

Another very useful way of looking at a use case specification is in terms of its start and end conditions. We can record all the possible start conditions that the system could be in; similarly, we can identify what state we expect the world to be in after we've finished. The use case steps, then, map the initial conditions to the appropriate final conditions.

Finally, we throw in constraints. A constraint is a requirement on not *what* the system does, but *how* it does it. A constraint may define a performance requirement, a reliability requirement, a safety requirement, etc. Each step in the use case specification can have one or more constraints applied to it. It is difficult to document how the constraint manifests itself in the use case but it is easier to document what happens when a constraint is not met. In this case the failure is treated as an exceptional condition.

My use case is not (necessarily) your use case

Since Jacobson defined use cases back in 1992 they have been subject to a vast range of interpretations. As Alistair Cockburn, author of *Writing Effective Use Cases* states:

> *"I have personally encountered over 18 different definitions of use case, given by different, each expert, teachers and consultants"*

I am no different: this is *my* personal interpretation of use case modelling and analysis. To qualify this statement though, this methodology is based on nearly a decade of requirements definition work on a number of high integrity projects, including military and aerospace.

That said, techniques that work in one environment may be less effective in another. I haven't been fortunate enough to work in every industry sector. It may be, then, that you disagree with some of my techniques. This is fine; I won't think of you as a bad person.

The 'rules'

Below is a set of guidelines, or rules-of-thumb. Each of the rules defines a good practice that I recommend when creating use cases. Each rule forms a quality control on the requirements analysis process. Ignoring a rule means there is an(other) opportunity for mistakes to creep into your requirements.

Ignoring these rules typically leads to wasted time and effort, low quality code, late projects, poor morale and a host of other project malaises.

This is not a complete set of rules. I have decided to focus on a manageable number. Ten would have been ideal; but that was too few and left out some important points. Twelve still fell short, so I decided on a 'baker's dozen' – 13 rules.

RULE 1: Use Cases Aren't Silver Bullets

There are a couple of popular misconceptions around use cases:

Misconception 1: The use cases are the requirements of the system

Contrary to what many engineers believe (and many authors have written) the complete set of use cases do NOT constitute its full set of requirements for the system.

A use case model is an analysis tool. They are a mechanism for organising the functional behaviour of the system and reflecting it back to the stakeholders. This is sometimes referred to as 'Problem Reframing'. By re-framing the requirements to are aiming to achieve three things:

- Demonstrate you have understood the problem, as the customer perceives it.
- Capture information exchange and sequencing requirements.
- Identify any missing behaviours

In order to achieve this effectively you need to generalise and abstract the customer requirements into something more manageable. Thus the use cases 'reflect' the system requirements without actually being the system requirements.

In embedded systems design the functional behaviour is but a small part of the requirements of the system. System developers must comply with a vast number of other requirements, including performance, reliability, security, environmental, useability, etc. Many of these are system qualities – that is, they apply to the system as a whole, not just the software. Use Cases are simply not an effective tool for capturing this information, despite attempts by several authors to incorporate them.

Misconception 2: You must always build a use case model

Engineering problems can be classified into four basic categories:

Data-oriented

In a data-oriented problem it is the information, and its relationship to other information, that is important.

Modal

Modal problems are characterised by having separate, distinct behaviours at different times. Trigger events from the environment will cause the system to change its behaviour.

Transactional

Transactional problems tend to be event-driven: A behaviour is (externally) invoked, which either produces a result or a change in the environment.

Flow-of-materials

Problems tend to be control-oriented: data/materiel moves from ‘sources’ to ‘sinks’. Algorithms and rules control how the information is moved and transformed.

While it’s perfectly correct to say that almost all systems have all these elements to some extent, in most cases one of the categories tends to dominate the requirements of the system.

Use cases are most effective when used to describe Transactional problems. Using Use Case analysis on other types of system often yields less-useful information about the system. In some cases Use Cases actually obfuscate the problem by attempting to re-frame one type of problem into a Transactional problem. For example, attempting to describe a flow-of-materials problem with use cases tends to yield trivial Use Cases and obscures the fundamental nature of the problem by trying to re-frame ‘flows’ as discrete ‘events’.

Use cases *are* a very powerful analysis tool – when used in the right way and under the right circumstances. But they aren’t a silver bullet. Use cases don’t solve every requirements analysis problem and they don’t necessarily suit every type of problem.

In order to use Use Cases effectively you must understand what type of problem you are trying to solve and whether use cases are the right tool for the job.

RULE 2: Understand your stakeholders

A Stakeholder is a person, or group of people, with a *vested* interest in your system. Vested means *they want something out of it* – a return on their investment. That may be money; it may be an easier life.

One of the keys to requirements analysis is understanding your stakeholders – *who* they are, *what* they are responsible for, *why* they want to use your system and *how* it will benefit them.

It's important to understand (and difficult for many software engineers to accept) your stakeholders have responsibilities above and beyond just using your product. In fact, the only reason they are using your product is because it (should!) help them fulfil their larger responsibilities. If your product doesn't help your stakeholder then why should they use it?

The first step in requirements analysis is to define your stakeholders. That definition must include:

- A named individual responsible for the stakeholder group
- The stakeholder's responsibilities. That is, a description of the roles, jobs, and tasks the stakeholders have to perform everyday. If you understand a stakeholder's problems and needs you can define solutions that help them
- Success criteria. That is: what is a good result for this stakeholder? The success criteria are a list of features and qualities that, if implemented, would bring maximum benefit to the stakeholder group.

Not all stakeholders are the same. For analysis, stakeholders can be broken down into three groups:

Users. The users directly interact with the system. The stakeholders are primarily concerned with functional behaviour and human-centred system qualities-of-service such as usability

Beneficiaries. The beneficiaries have some need that the system fulfils (or some pain that needs to be taken away!). The beneficiaries therefore benefit (often financially) by having the system in place. Typically, these stakeholders will be paying for the system. Beneficiaries are less interested in function and more interested in quality-of-service – reliability, maintainability, etc. since if these requirements are not fulfilled it will cost *them* money!

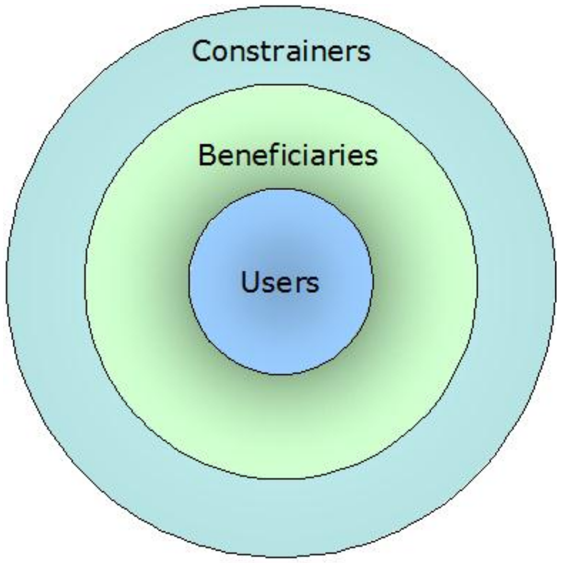


Figure 2 – The stakeholder 'onion' hierarchy

Constrainers. The constrainers place negative requirements – or design constraints – on the system. They place limits on how the system can work how it will be developed, or what technologies or methodologies may be used. Constrainers come in many forms – Legislation, Standards, The Laws of physics, to name a few. The development team itself is an important stakeholder, since it places limits on the technologies that can be implemented (lack of skills) or timescales (lack of resource).

Not only do the different stakeholders have different viewpoints, they also have different priorities on your project:

Beneficiaries' concerns typically (but not always) outweigh user concerns. For example, in the conflict between usability (a user concern) and low cost (a beneficiary concern) who will win? Remember: He who pays the piper calls the tune…

Constrainers should over-ride beneficiaries. Legal requirements, standards requirements, skills shortages, etc. will always supersede the desires of the other stakeholders.

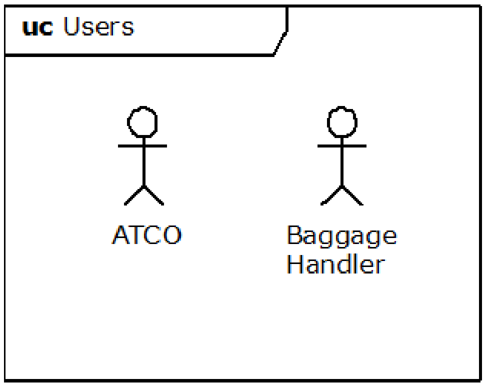
The core difference between Beneficiaries and Constrainers is that Constrainers CANNOT be influenced – that is, you can negotiate on functional behaviours or qualities of service, but you cannot negotiate away legal requirements or the laws of physics! A Constrainer either exists, in which case their criteria must be met; or they are not a Constrainer. The skill, therefore, is to reduce the number of Constrainers on a project to open up as many different design options as possible.

RULE 3: Never mix your Actors

The UML definition of an Actor is an external entity that interacts with the system under development. In other words: it's a stakeholder.

Having analysed all your stakeholders it's tempting to stick them (no pun intended) as actors on a use case diagram and start defining use cases for each.

Each set of stakeholders (Users, Beneficiaries or Constrainers) has its own set of concerns, language and concepts:



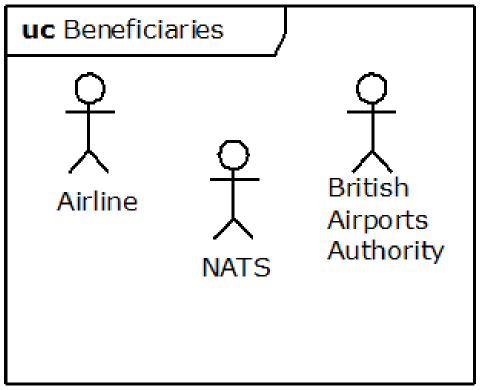


Figure 3 - - Keep actors from different stakeholders groups separate

Concerns. Each stakeholder group has a different set of issues, problems, wants and desires. For example, Users are interested in functionality; Constrainers in compliance.

Concepts. The way a system is perceived by the stakeholders depends on their viewpoint, their needs, their technical background, etc. Each group's paradigm – their way of perceiving the system – will be different and involve often subtly different concepts. For example, Users may have no concept of return-on-investment (RoI) for the system; whereas this may be a key concept to a Beneficiary.

Language. Just as concepts are different; so is the language used to describe them. In many cases, the same word is used in different contexts to mean different things. For example: how many different concepts of 'power' can you think of? Mechanical, physical, electrical, political…

It is vital never to mix actors from different stakeholder groups on the same use case diagram. Trying to mix actors leads to ambiguity and confusion; both for the writer and reader! The differences in concept, viewpoint and language will make the use case almost impossible to decipher and understand.

By all means draw a separate use case diagram for each set of stakeholders. (Note: non-User stakeholder use case descriptions is beyond the scope of this article)

FEABHAS

RULE 4: The "Famous Five" of Requirements Modelling

As I discussed in Rule 1, a common misunderstanding of use cases is that they are the software requirements. Unfortunately, this isn't the situation. Use cases are merely an analysis tool – albeit a very powerful tool (when used in the right situation).

Use cases are just one technique for understanding and analysing the requirements. In order to fully understand the requirements our use cases are going to need some support. Use cases are just one of my "Famous Five" of requirements analysis models.

The Requirements models are:

- The Use Case model
- The System Modes model
- The Context model
- The Domain model
- The Behaviour model

Why five models? Well, each one tells me about a different aspect of the system. No one point of view can tell me everything I need to know in order to ensure my requirements are coherent, consistent and unambiguous.

The **Use Case model** is focussed on interaction behaviour: the who, how, what and when of interaction between the stakeholders and the system.

Use cases focus on operational scenarios. For some systems (especially a lot of application software) the (transactional) exchange of information between the users, or other direct interactors, and the system forms the bulk of the software functional requirements.

However, many embedded systems are not user-centric, or transactional in their behaviour (for example, a closed-loop control mechanism). Anyone who has attempted use case analysis on such systems tends to find the use case model is non-intuitive to construct; and tends not to yield very much information about the behaviour of the software.

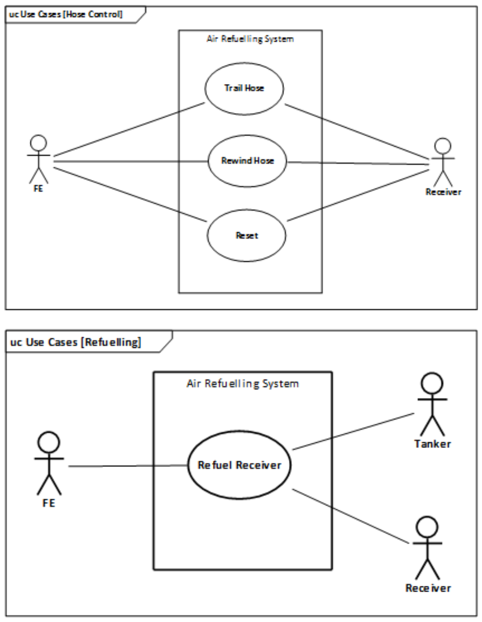


Figure 4 - The Use Case model

The **System Modes model** defines the temporal behaviour of the system. That is, how the behaviour of the system changes of time, in response to external (and internal) stimuli.

The System Modes model allows the analyst to capture when and how the system functionality is available. The System Modes model is a declarative diagram, showing the behavioural modes of the system (without saying how the behaviour will be enacted) and the signals or events that cause the behaviour to change.

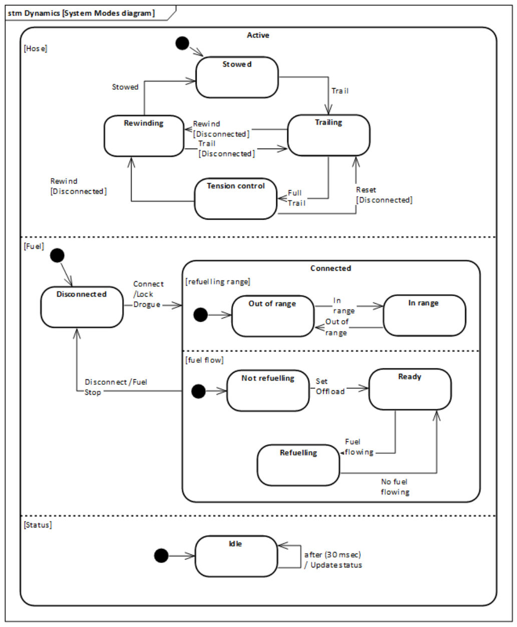


Figure 5 - The System Modes model

Application software may not be modal: it's either running or it's not. Embedded systems tend to have more complex dynamics (I see the system dynamics as one of the big differentiators between embedded software and application software). There are typically states where the system's primary functionality is available, and other states where it is not. For example, most embedded systems cannot provide their primary functionality when they are starting up, or shutting down, or in a maintenance mode.

The **Context model** defines the physical scope of the system: what is part of the system (under your control) and what is external to the system.

When creating requirements it is vital to separate the Problem Domain (the part of the real world where the computer is to exert effects) from the Solution Domain (the computer and its software). In fact, requirements should be describe in terms of the effects the system should exert in the Problem Domain (rather than how it should be designed). In addition there must be specifications for what are called Connecting Domains – that is, how the system's input/output devices must behave (interface specifications).

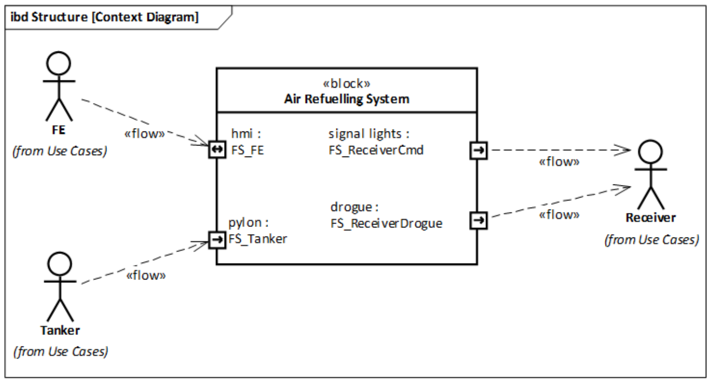


Figure 6 - The Context Model

The Context model gives a clear visual delineation of the Problem Domain (the environment), the Solution Domain and the Connecting Domains. The software is treated as a single black-box entity. The environment consists of the Direct Interactor stakeholders. Each Stakeholder interacts with the system via one or more interfaces (often called Terminators). For each element on the Context model there should be a set of requirements. In this example I have simulated a Context Model using a SysML Internal Block diagram.

The **Domain model** focuses on the information (that is, data) in the system and, more importantly, the relationship between the information.

The domain model aids with building a project 'glossary'. In any project there is a huge amount of tacit information about how the problem domain operates, and the language that is used to describe it.

The focus of the Domain model is understanding the problem and describing it, rather than specifying the problem's solution. Typically, a form of entity-relationship diagram is used. With UML, a class diagram is used (or a Block Definition Diagram in SysML).

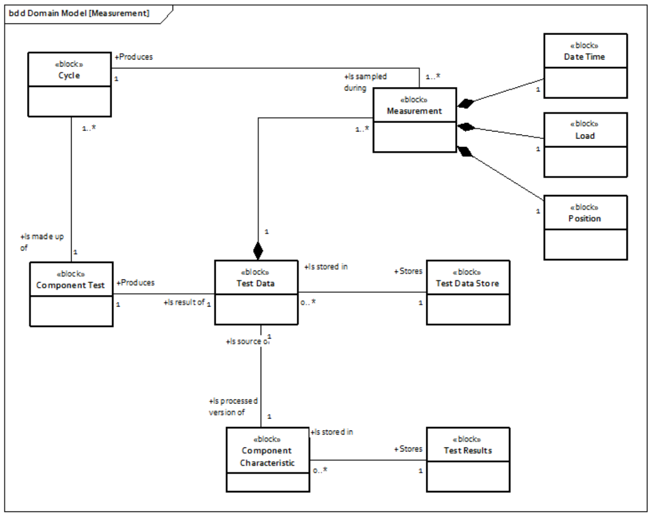


Figure 7 - The Domain model

It is tempting for development teams to skip this stage; the argument being "well, everybody knows this!" By actively and coherently modelling this information you may well avoid implicit misunderstandings; that can cost a project dear, if found too late.

The **Behavioural model** captures the transformational aspects of the problem. The Behavioural model focuses on sources and sinks of information, and what transformations are performed by the system in between.

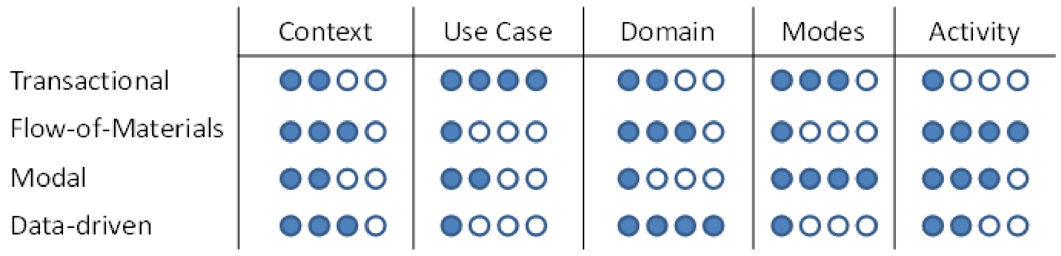
Although ultimately all software behaviour comes down to executing imperative code, this should be avoided for requirements analysis. Rather, focus on declarative statements of behaviour and where the data comes from (and goes to) rather than how the algorithms will be implemented.

The great strength of producing multiple models of the same system is that they are self-validating. Building a consistent set of models gives confidence that the analyst has truly understood the problem.

Concepts defined in one model must not conflict with the same concept in another model. For example, stakeholders defined in the Use Case model (its actors) must also appear on the Context model (otherwise, how are they interfacing to the system?!); similarly, the Use Case model should not mention any data or information that is not captured on the Domain model.

As I wrote in Rule 1, systems tend to have a predominant characteristic – that is, they will either be a Modal problem, a Transactional problem, a Flow-of-materials problem or a Data-Driven problem. When you are building your models of the system one or two diagrams will tend to give you more information than any of the others. For example, in a data-driven problem the Domain model will probably give you more information about the behaviour of the system than, say, the Modes model or Use Case model. The table below gives an indication of the relative value of each of the models.

Different models will have different value, depending on the type of problem.



	Context	Use Case	Domain	Modes	Activity
Transactional	●●○○	●●●●	●●○○	●●●○	●○○○
Flow-of-Materials	●●●○	●○○○	●●●○	●○○○	●●●●
Modal	●●○○	●●○○	●○○○	●●●●	●●●○
Data-driven	●●●○	●○○○	●●●●	●○○○	●●○○

Figure 8 - Different models have different value, depending on the nature of the problem

RULE 5: Focus on goals, not behaviour

There is a subtle distinction between the functional behaviour of the system and the goals of the actors. This can cause confusion: after all, the functional behaviour of the system must surely be the goal of the actor?

It is very common, then, for engineers to write use cases that define, and then describe, all the functions of the system. It is very tempting to simply re-organise the software requirements into functional areas, each one becoming a 'use case'. Paying lip-service to the 'rules' of use case modelling, these functions are organised by an actor that triggers the behaviour.

Use Cases based on functional requirements, rather than Actor goals.

I call these entities *Design Cases* to distinguish them from use cases; and they can be the first steps on the slippery slope of functional decomposition (see Rule 11 for more on this)

Identifying goals requires a change in mindset for the engineer. Instead of asking "What functions must the system perform?" and listing the resulting functionality, ask: "If the system provides this result, will this help the actor fulfil one (or more) of their responsibilities". If the answer to this question is 'yes' you've probably got a viable use case; if the answer's 'no', or you can't answer the question then you probably haven't fully understood your stakeholders or their responsibilities.

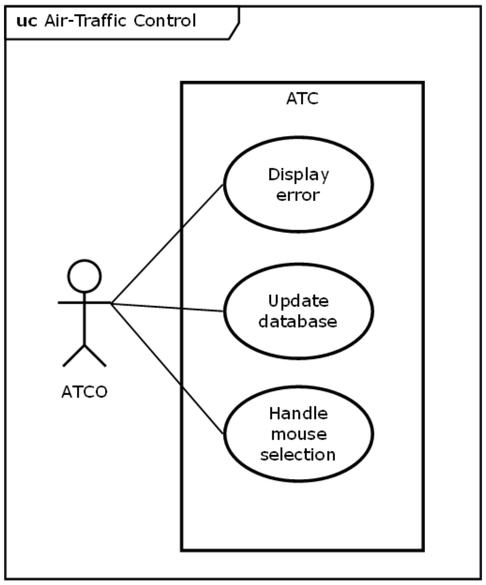


Figure 9 - Functional use cases; or "Design Cases"

In other words, the focus should be on the post-conditions of the use case – the state the system will be in after the use case has completed. If the post-condition state of the system can provide measurable benefit to the Actor then it is a valid use case.

Let's take a look at what I consider a better Use Case model (Figure 10).

The post-conditions of the use cases (above) relate to the goals of the Actor (in this case an Air-Traffic Control Officer). We can validate whether the post-conditions will be of value to the Actor.

The (main success) post-condition of *Land Aircraft* is that one of the ATCO's list of aircraft (that he is responsible for) is on the ground (safely, we assume!). At this point the aircraft is no longer the responsibility of the ATCO – one less thing for them to worry about. I argue that this is a condition that is of benefit to the ATCO.

Similarly with *Hand-off Aircraft*. As aircraft reach the limit of the local Air Traffic Control (ATC) centre they are 'handed-off' to another ATC centre; often a major routing centre. The post-condition for the hand-off will be that the departing aircraft will be (safely!) under the control of the other ATC, and removed from the local ATCO's set of aircraft he is responsible for.

Receive Aircraft is the opposite side of Hand-Off Aircraft. That is, what happens when the ATCO has an aircraft handed over to them from another ATC region. At the end of the use case, the ATCO must have complete details and control of the received aircraft.

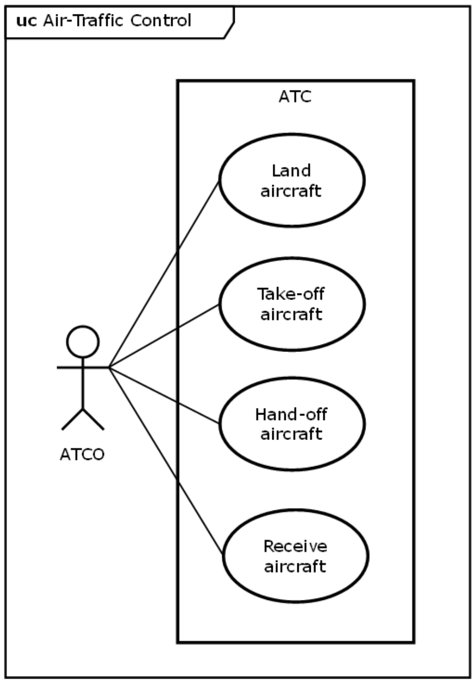


Figure 10 – Organizing use cases by actor goal

When an aircraft takes off, the aircraft must be assigned to an ATCO, who is responsible for routing it safely out of the local ATC region. The post-condition of *Take-Off Aircraft* must be that the aircraft is assigned to an ATCO and that ATCO has all required details of the aircraft's journey.

In the last two use cases, the ATCO actually gains work to do (another extra aircraft to monitor). The requirements of the system must ensure that when the new aircraft is received the transfer is performed as simply, or consistently, or straightforwardly, as possible. This is the benefit to the Actor.

While one could easily argue this is a simplistic model for Air Traffic Control it demonstrates basing use cases on goals rather than functional behaviours. Each use case is validated by its post-conditions, rather than its pre-conditions and behaviour.

RULE 6: If it's not on the Context or Domain models, you can't talk about it

Engineers love to solve problems. It's what they do. A use case model though is *not* a design model – it's an analysis model. Use cases describe what the system should do, and in what order. What use cases shouldn't do is say *how* the system should achieve these things. That's what design is for.

Stopping analysts (particularly if they're developers) from writing implementation details in the use case descriptions is difficult. One safe way of doing this is to limit the concepts written in the use case descriptions to only those defined on the Context or Domain models

Both the Context model and Domain model describe things beyond the scope of software implementation. That is, they describe the problem domain, not the solution (software) domain.

The Context model defines the physical parts of the system – external systems, users, interfaces, communication channels, etc.

The Domain model describes the informational context of the system – what artefacts exist, are produced, are inputs or outputs; and how these elements relate to each other.

All the data in these two models will exist irrespective of what software solution is developed. If they change then our understanding of the problem has changed.

When reviewing use cases look for concepts that are *not* defined on the Domain or Context models. These concepts are very likely to be implementation details. Look for items like 'database', 'CAN bus', 'Hardware Abstraction Layer', 'Observer', etc.

RULE 7: Describe ALL the transactions

A use case, as the name implies, describes the *usage* of the system from the point of view of some beneficiary (our Actor). This means the use case description must include the expected behaviour of the *actors* as well as the expected behaviour of the system. This can sometimes appear counter-intuitive to use case newcomers. If the use case is the *specification* of the system, then surely we can't impose requirements on our users (the actors)?

In fact, this is not the case. What the use case describes is – as stated above – the *expected* behaviour of the system. The use case describes what we (or, more correctly, our customers) would like to happen when we use the system. It describes the typical interactions between stakeholder and system, in order that the stakeholder achieves their goal. If the goal of the use case is beneficial to the stakeholder (see Rule 5) then the stakeholder has compelling reasons to behaviour as we describe in the use case. For example, if the system requests some information from the actor it is reasonable to expect them to respond with the information. In effect we are imposing requirements on the stakeholder ("when the system requests information, you shall respond"). Of course, there's nothing to stop the obtuse user from not responding, walking away, or doing any number of other bizarre and inexplicable things; but these behaviours are a fantastic source of exceptional flows.

The use case description defines the system behaviour as a sequence of *atomic transactions* between the actors and the system. Atomic is used in the sense of *indivisible*. This means that, *in the context of the problem domain* the transaction cannot be broken down into smaller transactions. See also: Rule 9 (below)

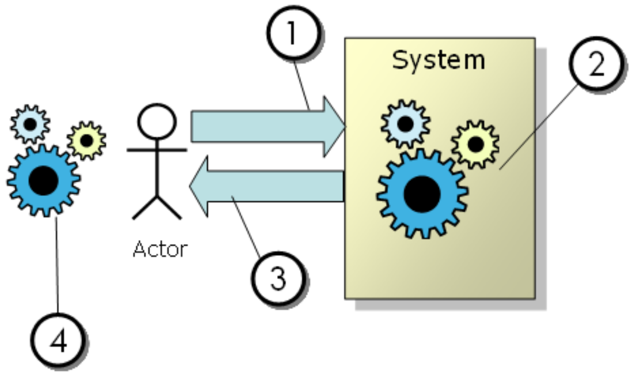


Figure 11 - Use case descriptions are a sequence of transactions between the actor and the system.

There are four basic transactions (See Figure 11) and the use case description will be made up of sequences of these transactions:

The actor supplies information to the system. Information could be an event (such as the trigger event) or data (or both).

The system does some work. Remember to describe the *results* of the work, not *how* the result is achieved

The system supplies information to the actor. Again, this could be requests for information, output signals, etc.

The actor does some work. As mentioned above this is unenforceable, but reasonable behaviour on the part of the actor.

Writing style

I suggest writing each transaction in a simple, semi-formal, declarative style. For transactions involving information exchange into or out of the system (that is, 'Actor supplies information'; 'The system supplies information to the actor') I use the following style:

[Source] [Action] [Object] [Target]

For example:	*"The Navigator supplies the Airfield Altitude to the NAV/WAS system"*
Source:	The Navigator
Action:	Supplies
Object:	Airfield Altitude
Target:	NAV/WAS System

For descriptions of work being done (either by the system or by the actor) I use the following form:

[Subject] [Action] [Object] [Constraint]

For example:	"*The ARS rewinds the hose at 5 ft/s +/- 0.5 ft/s*"
Subject:	The ARS
Action:	Rewinds
Object:	The Hose
Constraint:	5 ft/s +/-0.5 ft/s

If a transaction contains more than 3 punctuation marks it's probably too complicated and should be restructured to make its meaning better understood.

Weak phrases

Weak phrases make for comfortable reading of a use case but are apt to cause uncertainty and leave room for multiple interpretations.

Use of phrases such as "adequate" and "as appropriate" indicate that what is required is either defined elsewhere or, worse, that the requirement is open to subjective interpretation. Phrases such as "but not limited to" and "as a minimum" suggest that the requirement hasn't yet been fully defined.

Typical weak phrases include:

- As applicable
- As appropriate
- But not limited to…
- Effective
- Normal
- Timely
- Minimize
- Maximize
- Rapid
- User-friendly
- Easy
- Sufficient
- Adequate
- Quick

The truth is, in general engineers have poor writing skills. By providing a framework for how use cases should be written you are limiting the scope for ambiguity, wooliness and inconsistency.

RULE 8: Don’t describe the user interface

For many users the user interface *is* the system. Prototypes and mock-ups of the man-machine interface (MMI) are a fantastic way of eliciting requirements and use case behaviours from your stakeholders. And when defining use case descriptions adding MMI ‘screenshots’ and images can help illuminate the behaviour of the system to your customers.

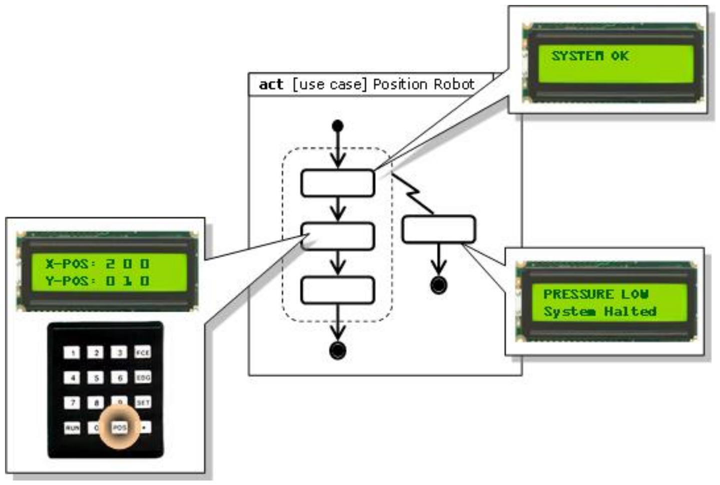


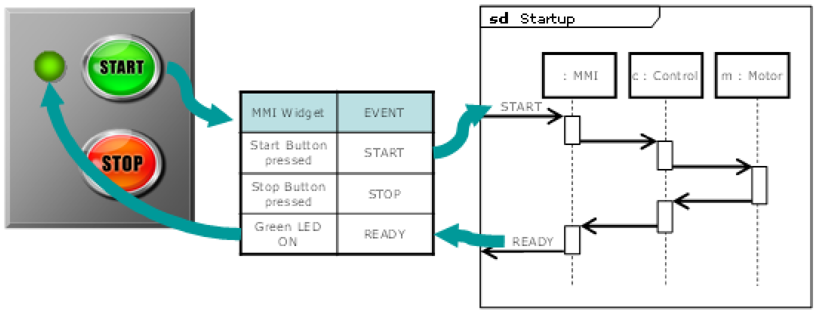
Figure 12 - Use the User Interface to drive the requirements analysis

It is very tempting to describe use case transactions in terms of MMI elements, in an attempt to make them more understandable to the customer. However, be aware: this is a very high maintenance solution. The one thing that can almost be guaranteed in the design of any system is that the MMI will change. A lot. By writing your use case descriptions in terms of user interface elements you will be constantly going back and revising your use case text (and reviewing, and doing impact analysis… you do these, don’t you?!)

To minimise the maintenance effort of a regularly-changing user interface we must uncouple the user interface from the functionality of the system. We must separate the effects of information flowing into and out of the system – What the information content is and how the system responds to it – from the presentation of that information to the user.

The use case description defines the desired requests and responses to the system; the MMI defines how those requests and response are manifested.

An event-mapping table can be used to map functions of the MMI (or indeed any interface specification) to the system event (transaction) it invokes or is in response to. Using an event-mapping table de-couples the interface from its function.



MMI Widget	EVENT
Start Button pressed	START
Stop Button pressed	STOP
Green LED ON	READY

Figure 13 - Using an event mapping to bind interface elements to system functionality

Events are typically one of the following:

COMMAND

A control input to the system
Used to control the behaviour of the system

INPUT

Normally represent a change in the environment
May be an analog or digital signal
The system may or may not respond to the change

OUTPUT

A signal to the environment from the system
Used to effect changes in the environment
May be an analogue or digital signal

STATUS

Feedback data from the system
Contains information about the current system state

The mapping table allocates some user interface operation (which could be as simple as a drop-list selection, or as complex as a whole sequence of button pushes or mouse clicks) onto a system behaviour.

In our simple example, pushing the big green button is our UI action. We've mapped this onto the '*START SYSTEM*' command. Similarly, when the system is ready, this '*READY*' status event is mapped onto the illumination of the LED. Now, if the user decides they don't want a 'ready' LED but instead want a 16 x 2 character display, we can simply re-map the '*READY*' status event onto a displayed message (something creative like "*System Ready*")

In this way you only need to change the use case description when the functionality of the system changes (hopefully, far less often than the MMI!)

Doing this separation also alleviates that all-to-common problem of poorly-written requirement specifications: the User Interface Specification contains half the functional behaviour of the system; and the System Requirements Specification contains half the user interface detail!

RULE 9: Build yourself a data dictionary

Transactions between the actors and the system typically involve the transfer of data. This data has to be defined somewhere. If you've built a Domain Model most of the data will be identified there; but even then the class diagram is not always the most practical place to capture the sort of information you need to record.

Another way to capture this information is in a *data dictionary*. This is a document that was *always* written a decade or so ago but seems to have gone out of fashion in many software development circles.

A data dictionary defines each piece of data in the system, and attributes about that data. Typically, a data dictionary holds (but is not limited to) the following:

An Identifier. This is how the data will be referred to. Remember to use a term that has meaning in the *problem* domain.

Definition. What does this data represent? (And remember: a good definition consists of a *genus* – what *type of thing* I'm defining – and a description)

Units of measure; if applicable

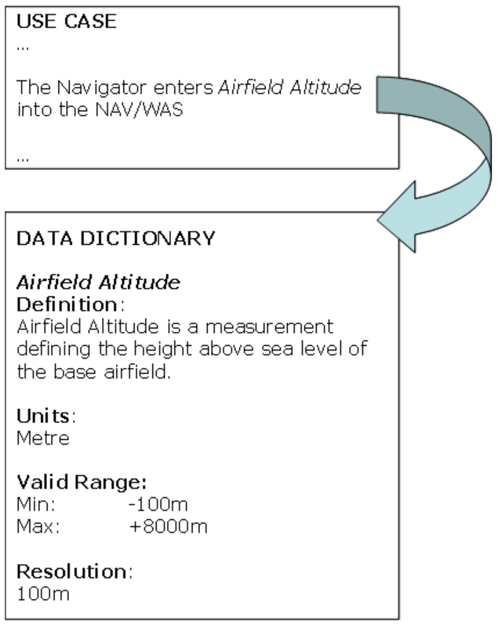
Valid range; again, if applicable

Resolution. That is, what's the smallest difference I can measure? This can become important if your implementation may use fixed point number schemes (for example).

This information is all vital to the developers who must design and code interfaces, etc. but it also has benefit when writing use case descriptions – it decouples the behaviour of the system from its data.

Just as we should decouple user interface details from the use case description (see Rule 8) so should be decouple the data transactions. Describing the data requirements of a transaction in the use case description is cumbersome and leads to a potential maintenance problem (what if the resolution of a data item changes? Or its units of measure? Ask the team that developed the Mars Climate Orbiter about getting units of measure confused!)

As you define your data dictionary, you can refer to data items (by their name) in the use case text. Use


USE CASE
...

The Navigator enters *Airfield Altitude* into the NAV/WAS

...

DATA DICTIONARY

Airfield Altitude
Definition:
Airfield Altitude is a measurement defining the height above sea level of the base airfield.

Units:
Metre

Valid Range:
Min: -100m
Max: +8000m

Resolution:
100m

Figure 14 - Use a data dictionary to de-couple data items from use case descriptions

some typographical convention (I use italics) to identify a data item from the data dictionary; or, you could hyperlink it.

If the data requirements change you shouldn't need to change the use case – unless it leads to a change in behaviour. Similarly, you don't need to elaborately detail exception flows for invalid data. The data dictionary defines *what is valid*; the use case defines *what should happen* if we receive invalid data. A simple, clear separation of responsibilities.

RULE 10: The magical number seven, plus or minus two

Psychologist George Miller, in his seminal 1956 paper *"The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information"*, identified a limit on the capacity of human working memory. He found that adults have the capability to hold between five and nine 'chunks' of information at any one time. A 'chunk' may be a number, letter, word or some other cohesive set of data.

What has this to do with use cases? One of the primary functions of writing use cases is requirements validation – that is, are we actually building the correct system? The use case model is a technique for presenting the system requirements such that the customer can say either yes, we have a correct understanding of how the system should work; or no, we have misunderstood the system.

When presenting information to our (probably non-technical) customer it makes sense to keep the information content manageable. Try to keep the number of transactions in any one flow (that is, a sequence of use case steps between a trigger event and a conclusion) to between five and nine. This gives your customers the best chance of comprehending what you're writing.

This means that the use case stops being a list of *all* the system requirements and becomes a *précis* of the system requirements. Each transaction may therefore equate to more than one system requirement. In practice this is not as over-simplifying as it sounds: requirements are often 'clustered' around elements of data and their production and manipulation – effectively what each use case transaction describes.

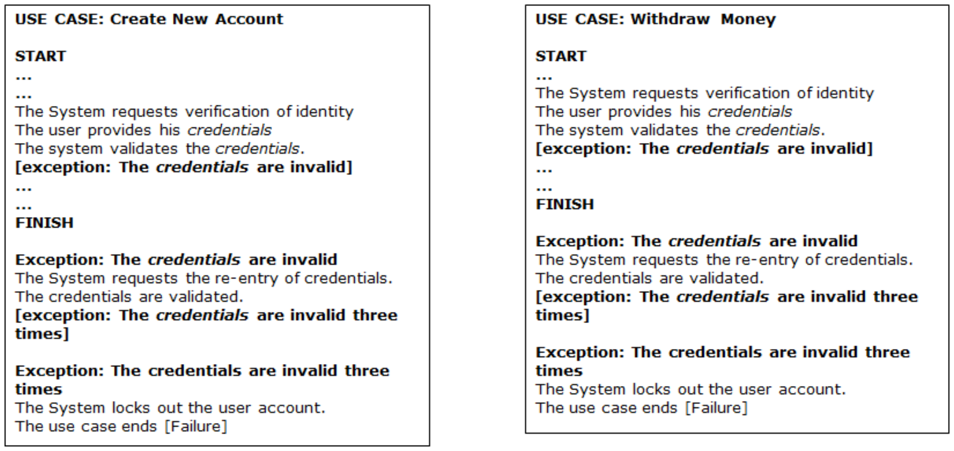
The skill in writing good use cases is the ability to précis requirements together to minimise the number of use case steps, without creating simplistic use cases.

The 'seven, plus or minus two' rule is not hard and fast; more a guideline. If your description has ten steps, this is not a problem; however, if it has thirty then there's a chance you've over-complicated the step. The same is true at the other limit: three steps is fine; one step means you *may* have over-simplified and lost detail.

RULE 11: Don't abuse <<include>>

A use case contains all the steps (transactions) needed to describe how the actor (our stakeholder) achieves their goal (or doesn't; depending on the particular conditions of the scenario). Therefore a use case is a stand-alone entity – it encapsulates all the behaviour necessary to describe all the possible scenarios connected to achieving a particular end result. That's what makes use cases such a powerful analysis tool – they give the system's requirements context. Use case are also an extremely useful project management tool. By implementing a single use case you can deliver something complete, and of value, to the customer. The system may do nothing else, but at least the customer can solve one problem with it.

Occasionally, two use cases contain a sequence of transactions common to both sets of scenarios. This sequence may not necessarily occur at the same point in each use case (for example, the beginning) but will always be in the same order.



USE CASE: Create New Account

START
...
...
The System requests verification of identity
The user provides his *credentials*
The system validates the *credentials*.
[exception: The *credentials* are invalid]
...
...
FINISH

Exception: The *credentials* are invalid
The System requests the re-entry of credentials.
The credentials are validated.
[exception: The *credentials* are invalid three times]

Exception: The credentials are invalid three times
The System locks out the user account.
The use case ends [Failure]

USE CASE: Withdraw Money

START
...
The System requests verification of identity
The user provides his *credentials*
The system validates the *credentials*.
[exception: The *credentials* are invalid]
...
...
FINISH

Exception: The *credentials* are invalid
The System requests the re-entry of credentials.
The credentials are validated.
[exception: The *credentials* are invalid three times]

Exception: The credentials are invalid three times
The System locks out the user account.
The use case ends [Failure]

Figure 15 - Use cases sharing common behaviour

UML provides a mechanism for extracting this common information into its own use case. The mechanism is called the «include» relationship (Figure 16). The semantics of the «include» relationship mean that the base use cases are only complete if they fully, and completely, contain the contents of the included use case.

This relationship can sometimes be useful, particularly if a sequence of transactions is repeated many times.

However, misunderstanding of «include» tends to lead to a very common abuse: functional decomposition of use cases.

Many use case modellers use the «include» relationship to split a complex use case into a number of smaller, simpler use cases (see Figure 17). This can lead, in extreme cases, to an explosion of use cases, with leaf-node use cases capturing trivial functional requirements (for example "Capture button press").

These trivial use cases often have no meaning to stakeholders, who should be focussed on what they want to happen, rather than how it happens. Also, there is a huge overhead is creating, reviewing and maintaining this vast number of use cases.

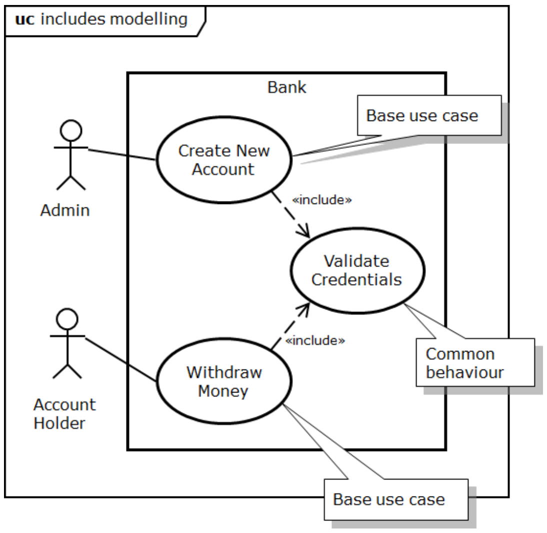


Figure 16 - Included use case notation (if you *must* use them)

This effect is a typical symptom of functionally-oriented use cases, or '*Design Cases'* as I call them.

Rather worryingly, several organisations actively promote Design Cases for requirements analysis. The appeal is obvious: functional decomposition is a familiar concept with most developers (and if it isn't they really shouldn't be developing software!); and it allows the developer to settle back into the comfortable territory of solving problems (rather than defining them)

Remember, use cases are an analysis tool for understanding the system. A simple, coherent set of use cases, reflecting the usage of the system from the customer's perspective is far more effective than demonstrating, to the n-th level, how the system will operate. That's what design verification is for.

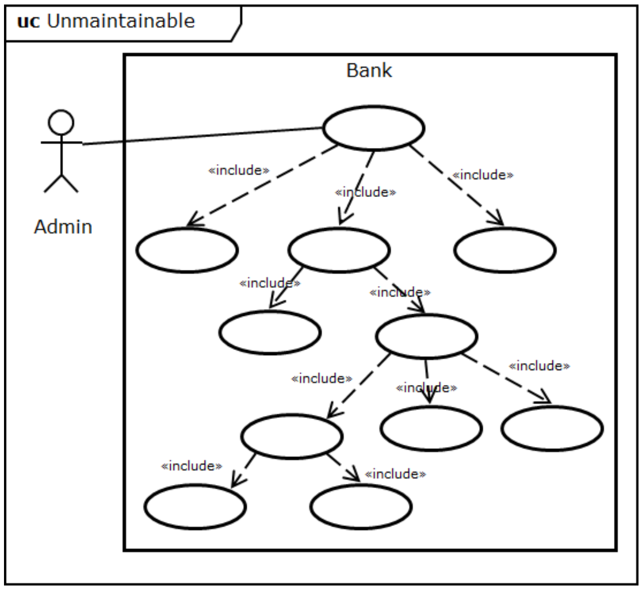


Figure 17 - Design cases in action!

My advice is to avoid «include» wherever possible. Prefer repetition of text within the use case description, over trying to identify and extract commonality. In this way each use case remains separate and complete; and there is no temptation to fall into the functional decomposition trap. That way lies madness (or at least analysis paralysis).

RULE 12: Avoid variations on a theme

A common affliction amongst novice use case modellers (particularly those from a development background) is the desire to fettle the use case model – to organise it, revise it, balance it; and generally make it look more like a design model. Unfortunately, beyond a certain point this effort actually starts to degrade the utility and effectiveness of the model. More and more effort is put into a model that becomes less and less useful to the customer and the analyst.

This all-too-common situation is known as 'Analysis Paralysis'.

I always advise, unless you can provide a really compelling reason to do otherwise avoid associations between use cases. Yes, you might have a less optimal model, but at least your stakeholders will find it easy (or easier) to follow. This is price you always pay with use case associations: Elegant but complicated to understand, versus sub-optimal but explicit. In my view explicit always wins; after all, we're trying to understand the problem, not design a solution.

Case in point: Use case specialisation.

In UML a use case (and indeed an actor) is a special form of classifier. One of the properties of classifiers is that they can be specialised – that is, a new classifier may be defined that has the properties and behaviours of the parent, but may extend or modify these attributes. The definition of the specialised classifier is that it is also a the type of its parent – that is, all instances of the specialised classifier form a subset of the set of parent classifiers.

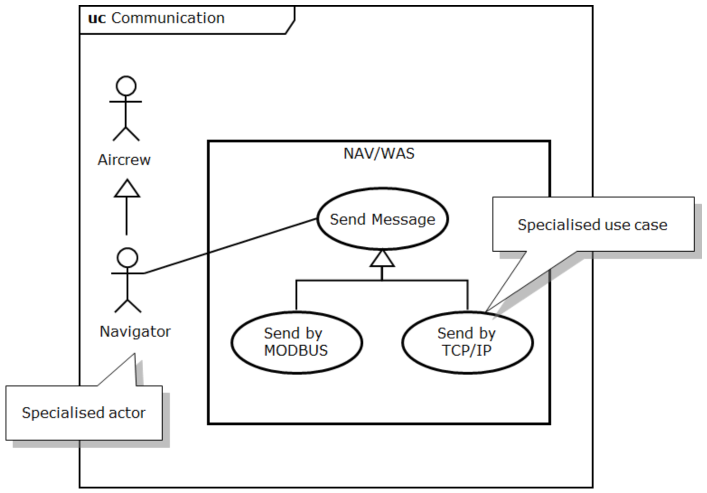


Figure 18 - Use case and Actor specialization

In use case terms it means we can define actors or use cases that represent specialisations of some base behaviour. As part of our analysis we may choose to define actors whose goals form a superset of another actor's. For example, we could define an Administrator actor as a specialisation of a User actor. The Administrator performs all the functions of the User but has additional behaviours (admin tasks)

Actor specialisation is avoidable if you consider actors as roles. A role is just a set of competencies and behaviours that can be adopted by a person (when interacting with the system). So, when I perform everyday tasks I adopt the role of a User actor; when I need to do an administrative task I adopt the role of an Administrator (if I'm able) This model also means I can be an Administrator without being a User – something the specialisation model doesn't allow.

For use cases the problem becomes more complex. Given any particular use case, what does it mean to specialise it? A specialised use case is like a base use case, but may have different behaviour. This complication is adding nothing to our comprehension of the system (in fact, possibly the opposite!). Not to mention the pain you will go through trying to explain all this to your customers!

The **Liskow Substitutabiliy Principle** (LSP) can help with this situation. In simple terms the LSP says that a derived class can be substituted for a base class object if, and only if, the derived class has no stronger pre-conditions (demands no more of the client) and has no weaker post-conditions (delivers no less).

For use cases, this means the specialised use case must have the same pre-conditions as the super use case (or possibly weaker). The specialised use case may override super use case functionality (whatever that actually implies!). Finally, none of the scenarios in the specialised use case must result in weaker post-conditions than the super use case.

This could be achieved by having a super use case with limited (or no) behaviour, strong pre-conditions and very weak post-conditions.

However, one could reasonably argue this super use case becomes little more than a generic place-holder and doesn't really add much to our understanding of the system's (transactional) behaviour.

Use case specialisation can be avoided very simply by forking the use case with an optional flow. For one variation take one path through the use case; for another variation take the other. Unfortunately, this causes the use case to become more complicated; but at the benefit of explicitness.

There are some rare cases where use case specialisation can be more explicit but these are more the exception than the rule.

RULE 13: Say it with more than words

Use case descriptions are most commonly written in text format (albeit often a stylised, semi-formal style of writing). Text is a very effective method of describing the transactional behaviour of use cases – it's readily understandable without special training; most engineers can produce it (although the ability to write basic prose does seem beyond the capability of many!); and it is flexible enough to deal with complex behaviours – for example, variable numbers of iterations through a loop – without becoming cumbersome.

However, this flexibility can come at the price of precision. Sometimes, for example in many control systems, you have to state precisely what the response of the system will be. Words, in this case, are not adequate – although I have seen requirements engineers attempt to describe the overshoot behaviour of a closed loop control system in English prose!

This rule therefore should be: find the *most appropriate* notation to describe the transactions between the actors and the system; with a corollary that sometimes the right notation is *multiple notations.* Use tables, charts, diagrams, activity diagrams, flowcharts, mathematical models, state charts, or combinations of these, to describe the behaviour. Don't get caught up in the dogma of use cases that says the use case description *must* be text. For most systems I have found that combinations of methods work best – some transactions are best described using semi-formal text; some by state machines; others by tables. The skill is recognising the type of transactions you are describing and picking the appropriate method.

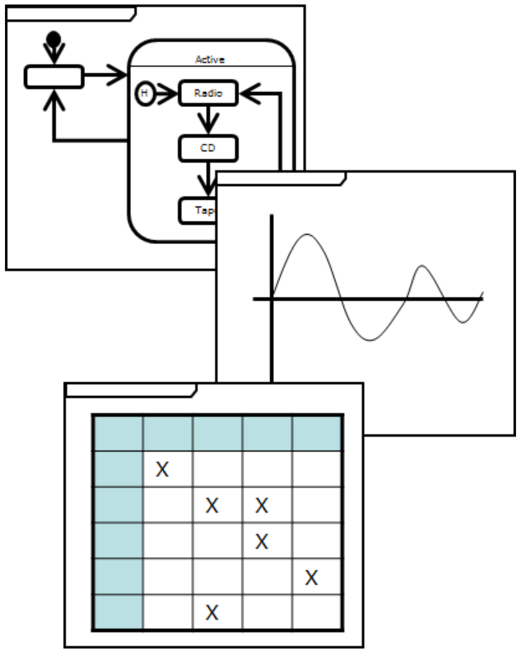


Figure 19 - Use the most appropriate notation for what you are trying to describe

END OF DOCUMENT