

1.3 Requirements Engineering

Large Government IT projects suffer from late delivery and from cost over-runs. Part of the problem lies in the way these projects are purchased (procured) which has institutionalised a flawed process, largely because of the imprecision with which requirements are specified. The prices that are quoted on tender documents rarely bear any relationship to the final outcome because they are calculated in an informal way, that may be more oriented to winning the business (with an inexpensive quote) than with being accurate. This section investigates what initiatives have been undertaken to make requirements' expression less ambiguous.

Requirements engineering (RE) is a relatively new branch of software engineering. As the name suggests it is concerned with the capture and representation of user requirements. Although RE was identified in 1976 at the International Conference on Software Engineering, the first conference devoted to RE was not held until 1994 in Colorado Springs. It was at this conference that the subject of RE was first addressed 'holistically' [1]. Since then the subject has grown in both importance and status, as the realisation has emerged that the biggest threat to software project delivery lies in the inadequacy of requirements' expression. RE may be the activity with the highest return on investment of all software engineering tasks. This can be claimed because it is agreed that it is much cheaper to rectify errors in requirements than it is to do so during software testing by an estimated factor of 200-1 [2]. There might be no requirements issues were there no cost constraints, and customers were sufficiently motivated and talented to specify accurately and completely [1]; a utopia. In this section a brief survey of the literature in RE is offered and discussed, before a fuller treatment of use case modelling is presented.

What is meant by a 'requirement' in the context of software engineering? A requirement is a statement of the real need of the customer; it is testable, and it may be prioritized. Requirements must be stated in an objective manner, as they are more than a vague notion of a customer 'need'. A requirement is the output of the elicitation process that has a specific meaning to a software engineer [1]. Zave offers a classification of RE effort. RE is the branch of software engineering (SE) concerned with real-world goals, constraints, and precise specifications of software behaviour. The subject is broad, interdisciplinary and open-ended.

Sometimes it takes the form of translation from informal observations of the real world into mathematical specification languages [3].

RE is a broad field in which there are many opportunities to make a contribution. Requirements are used as direction to project management by providing a theoretical underpinning to that activity. When the constituent themes of RE are examined, it is clear their collective goal is to improve the chances of project delivery success. RE embraces barriers to communication, the conversion of vague goals into specific behaviour, understanding priorities and measurement of satisfaction while simultaneously developing strategies for allocating requirements amongst components that will satisfy them. The subject includes research into estimating costs, risks and schedules. It is concerned with ensuring completeness, and the integration of multiple views and representations. The task of obtaining complete, consistent and unambiguous specifications falls under this umbrella, as do problems of managing evolutionary change. Even patterns can be considered part of RE, as both are concerned to encourage the reuse of artefacts for the future development of similar systems.

This is a 'baseline' definition that describes the constituent parts of RE to which other researchers have contributed [4]. The importance of case study reports has been recognised to RE as they provide important evidence. Of necessity, the evidence of case studies is anecdotal but van Lamsweerde *et al* considers this an inevitable precursor to the definition and execution of experiments that may be subjected to more systematic evaluation [5]. It has been suggested that the ultimate objective of RE should be to undertake the evaluation and comparison of proposed solutions within a controlled experiment capable of quantitative measurement [6].

Requirements' expression can be undertaken using natural language, formal language, or with semi-formal constructs. There is no substantial body of research in the literature on natural language expression apart from to point to its inadequacy (*see: Natural language on page 15*), although there are guidelines that are believed to distinguish between better and lesser representation [7]. Formal language expression, as characterized by formal language theories (e.g. 'Z') [8] have a role to play, albeit a limited role [9]. Examples of semi-formal constructs are represented by the use case notation that forms part of the UML [10].

Prior to the acceptance of RE as a distinct function, the task of requirements gathering was undertaken in the *analysis* function. The change from an emphasis on analysis to an emphasis on RE has taken place because it was necessary to make the distinction between problem elicitation and solution discovery. Analysis, as a task, has been considered part of the solution space.

One of the objectives of analysis was to identify persistent entities that will be represented as data entities. The simple principle that is applied to this task is based on natural language grammar; specifically the analysis of nouns. As far back as 1983, Abbott noted the striking parallel between noun phrases in an original text and the objects that would later become abstract data types also realised as database tables [7]. Abbott does not suggest that all the nouns in a problem statement should be transformed into classes in software, instead he introduces the concept of a ‘common’ noun which will become a class, whereas a proper noun is an example (or an instance, or an object; in this sense all synonyms). The approach is simple in concept, but has defied automation as the task requires human intervention to arbitrate in the area of language usage and context. Furthermore, Abbott’s work goes on to characterise verbs as being realised in code as procedural or method calls.

Requirements Management

Requirements change during and after development. 70% of projects may be at risk due to requirements volatility [11]. Changing requirements have been termed ‘scope creep’, but some ‘new’ requirements may, in fact, simply be adding more detail to what has already been captured, albeit on a lower level of goal abstraction (*see*: Abstraction on page 36).

A budget and plan is only valid against an agreed specification. As the specification changes, the plan will change. Berry compares house building with software engineering. In house building it is common for a supplier to charge inflated prices for changes as he will try to recover from bidding a low (uneconomic) bid in the first instance. The differences Berry identifies between the two domains are that software engineers try to incorporate changes without charging for them, or adjusting the schedule due to misguided motives. The feeling may exist that changes to software are easy because software is ‘flexible’, whereas builders

have no qualms about presenting bills for ‘extras’ [12]. Although there are tools that can help with requirements management such as Doors or Requisite Pro¹, the old software engineering adage ‘garbage in, garbage out’ remains true; tools will not help when the content is wrong. The management of requirements is a prerequisite of any organisation hoping to attain level 2 of the Software Engineering Institute’s Capability Maturity Model (CMM) [13] which demonstrates an organisation has a repeatable process.

The concept of ‘goals’ is central to the study of RE [14]. This is in contrast with methodologies that treat requirements as consisting only of processes and data (traditional systems analysis) such as that described by DeMarco [15]. The objective should be to focus on understanding and characterising the interaction between the intended system and its environment and not simply to become fixated on what the system is supposed to do [16].

Hierarchies of Abstraction in Requirements Engineering

It is part of the orthodoxy in software engineering that the solutions an organisation builds in the future will be too different from the solutions they have built in the past that there is little hope of being able to profit from experience in the same way that civil or electronic engineers are able to do. Two interesting recent contributions have challenged this view, with the potential for far reaching changes to the world of software engineering. In the not too distant future it may become a quaint old-fashioned notion that software projects are really so very different from one another. By way of clarification, it is certainly true that systems that have been built *differ*, but what authors are now postulating is that the problems they address are, in essence, similar, and perhaps even more surprising, are also simple in nature. This is a radical statement that flies in the face of the software engineering orthodoxy, and may lead to a undermining of the reasons that have so often been offered up for the dismal record of software project failure. The work that has been done to justify this new position, that software problems are similar, that there only exist a few different types and that they are fundamentally simple, lies in the introduction of a new dimension of abstraction; the abstraction of software problems, as distinct from software solutions. The two authors who have suggested

¹ Doors is a product of Telelogic. Requisite Pro is a product of Rational Corporation, which is now owned by IBM.

this are Michael Jackson in his book Problem Frames [17] and Alistair Sutcliffe [18] with significant input from Neil Maiden [19]. These authors are at the forefront of thinking in RE.

Problem Frames

Jackson recognises that a solution-oriented approach may work well where the problem being tackled is well known and classified. He believes however, that for most software problems this is not the case, because there is no system for the general classification of problems. Rather heroically, he then goes on to define one [17]. In this view of the world, problems are constructed from entities, events, values, states, truths and roles that exist within a problem domain. There exists the machine domain which consists of both the computer and the software that depends on it. In order for a solution to be constructed, the two domains of the machine and the problem must share *phenomena*. Requirements, as understood in this context, are properties of the shared domain but which are not possessed intrinsically. It is the task of the designer to endow the machine with the properties necessary to satisfy the requirements.

A set of reusable standard elementary problem frames (PF) are introduced that capture characteristics of a specific problem from an idealised class. High-level problem classes correspond to intuitive notions of different kinds of problem but make the intuition more precise. They stipulate the structure and characteristics of the problem domain. These elementary frames form the basis of the technique. In themselves, simple frames are too atomic to represent real world problems, but they are capable of being combined into composites. Large problems are composites of small problems. Small problems must be recognised problem classes that can be described in the notation of problems frames. Jackson's thesis is that the world of problems can be profitably represented by a rich set of reusable composites. In the Jackson world view, problems are in the real world, not in the computer. The computer is where the solution resides. The art of building a good solution begins with recognition of the basic structure of the problem through the identification of the applicable problem frame composite. To apply the approach, each big problem must be de-composed into sub-problems that can be fit to simple frames that are subsequently combined. This focus on problems concentrates the mind of the practitioner with the intent of avoiding 'drift' into the contemplation of solutions too early.

Problem frames share much of the spirit of design patterns however, whereas a design pattern looks inward toward software, a problem frame looks outward to where the problem exists. What they share, is an approach that encourages the identification and reuse of knowledge that can be applied to recurring situations. The practitioner has available a taxonomy of experience and knowledge which exist in a layered schema that can be shared and effectively accessed. Where an architectural question might be solved by recognising the requirement to apply a *decorator* pattern a particular RFP might equally be identified as representing a *workpiece* frame problem.

Jackson says that “if you want to understand anything, you musn’t try to understand everything.” From this he means to imply that problem frames are applicable to the identification of recurring classes of problem that can be solved with software solutions, however, they do not have any social, political, organisational or economic dimension. For instance, they are of no help in the elaboration of a business case for why a particular project should go ahead. Further, he asks what the value is of being vague when it is possible to be exact; he believes that it is preferable to be as formal as the project or task demands.

Another profound opinion Jackson expresses is that no experiment should be ignored simply because it has been labelled a ‘toy’ problem. Such a criticism implies that reported results could not be replicated on a bigger project. Jackson specifically states that “[it is wrong to] spurn trivial problems. They show the stripped-down essence of problem classes in their barest form that makes it much easier for you to recognise them when they appear in fancy dress concealed by the trappings of a larger problem.”

Practitioners have a reputation for solving the wrong problem. Jackson asserts it is crucial to look behind what the customer says, to interpret what they really mean, and to feed back to them a vision of what they ought to be saying. He is critical that, as software professionals, there is contentment to solve superficial problems without tackling the real problems that reside underneath.

Jackson identifies the elementary frames as *required behaviour*, *commanded behaviour*, *information display* and *simple workpiece*. *Required behaviour* describes physical objects that require control whereas *commanded behaviour* describes control based on instructions. These two PFs are helpful in describing real time and embedded software applications. They are not considered further in this thesis and the interested reader is referred to the source material.

A *simple workpiece* frame describes the creation and editing of structures; in this example, data structures. The machine in the frame can copy, print, and analyse a class of computer 'processable' text. A *workpiece* is defined as a material worked on by a machine or tool. The machine controls the event phenomena. Events are the operations on the *workpieces* by which they can cause a change in symbolic value and state. The frame allows for the machine to examine current state and values of the *workpiece*. The *workpiece* domain itself is considered inert. Changes in state are in response to externally triggered events. No state changes are self-initiated; they are controlled by the user via issued commands. Some commands may be disobeyed as they violate defined rules. Display and printing are not handled by this frame; they are defined by the *information display* frame. Essentially, the simple *workpiece* frame defines behaviour over a lexical domain, as it stores symbolic phenomena and allows for later retrieval. Additionally, the information display frame displays information but does not allow for any storage. These two frames work commonly together, and together they form the composite, 'interactive *workpiece*' frame. Sutcliffe characterises the '*workpiece* frame' as including editing operations such as create, change, delete and view; a set that has elsewhere been defined as the CRUD set (*see*: CRUD functionality on page 33).

Jackson says that "an essential part of mastery is focusing your effort and attention on what's important. In software development the touchstone of effort and attention is description." Problem Frames [17] is a major contribution to the RE lexicon. Instead of describing what is wrong with SE practice and bemoaning the inherent complexity that renders improvement unlikely or at best incremental at a frustratingly slow speed, he advocates that this obsession with the contemplation of failure holds back improvement. In Jackson's opinion the industry's ability to solve problems has been too shallow.

Jackson advocates never losing sight of the fact that it is a model that is being created, as opposed to an accurate and complete picture of the world. A model deliberately ignores some aspects of the real world because the description needs to be compact and available in a single document, as opposed to being scattered amongst many documents. For this reason, he states that it will never be possible to describe an entire system with use cases (*see*: Use Case Modelling on page 16); they can never be sufficient as use cases represent only fragments of behaviour.

Domain Theory

Domain theory (DT) [18] is an ambitious idea that states that all software engineering problems can be described by a small set of models that represent fundamental abstractions. It is a complimentary work to that of Jackson [17]. Abstractions are identifiable in the world of problems and their identification is intended to act as a counter balance to the abstractions of patterns that describe design solutions. Experts form abstract models. It is the act of recognising something similar about the problem that gets experts started on proposing efficient solutions. This can be described as the “I’ve come across this problem before” feeling [18]. Theories, ideally, should explain and predict phenomena in the world. Problem abstraction is important because it forms the entry point to solutions.

To Sutcliffe an application domain is a sector of industry where requirements are found. He proposes a set of problem oriented abstractions where, following Jackson, the emphasis is on problem description as distinct from solution description.

Reuse is one of the holy grails of software engineering, whereby existing artefacts are used again to construct new systems. The emphasis has been on the reuse of solutions, and although the theory of reusability is uncontroversial, it has not been achieved on any systematic level. The basic motivation behind a strategy of reuse is to save time and effort while concurrently improving quality. True reuse implies some conscious design of the module for this purpose [20]. There has been little reuse of components beyond the level of system software. The cause has been held back, because unless the problem is properly articulated and classified, it is not clear how the solution may be identified.

Generalisation is linked to the trade-off between utility and the scope for reuse. A generalised component is an abstract component, which by definition contains less detail. Here lies what may be termed a ‘utility trap’, where a component may be applicable to many domains but offers little benefit to any without substantial customisation. Therefore, less detail implies potential in a wide spectrum of domains, at the penalty of decreased immediate utility. Reuse must be predicated on some over-arching theory of partition that predicts how components should be specified while delivering sufficient opportunity for customisation that avoids the trap of low utility.

Reuse contains a wider dimension that may seek to capture concepts at different levels of abstraction ranging from wisdom and knowledge, to behaviour and hierarchies of data. Psychologists might characterise the process as the reuse of information by processing it into higher order abstractions called ‘chunks’ [21]. With a sufficiently robust collection of layered ‘chunks’ the structure may come to constitute a knowledge hierarchy (or graph) that can be traversed. Elements assembled for the purpose of proposing a solution to an identified problem can be found. In all cases, successful reuse is predicated on thorough identification of the problem.

According to Sutcliffe software applications can be considered to inhabit a vertical hierarchical structure, shown in Table 1.3.1, which defines the relative ease of establishing reusability.

Table 1.3.1: Hierarchy of software types according to Domain Theory.

Layer	Class	Examples
1	business	banking, health, education, avionics
2	tool	word processor, spreadsheet
3	component	classes, patterns
4	infrastructure	database, network, system

According to Sutcliffe, the lower the level of the hierarchy the more liable software is to be successfully reused. Infrastructure layer software, i.e. operating systems, are commonly reused, whereas business layer software i.e. banking applications, are less so. COTS products tend to target software in the business layer and rely on customisation through configurable parameters.

‘Design by reuse’ is a matter of capturing requirements in an unambiguous manner, and seeking to fulfil them through the creation of a software application. Applications may be built from ‘scratch’ (bespoke development), or through the assembly of reusable components as in the specialisation of a COTS application. If the component assembly route is selected, inevitably existing components will not exactly match new requirements. One way to tackle this ‘mis-fit’ is to build easily customisable components through providing a mechanism of parameterisation that has the effect of changing application behaviour without resorting to writing new code.

Sutcliffe views the power of abstraction in its ability to act as a bridge between requirements, design and software realisation. He identifies the failure of reuse on the inability of users and designers to share the same language of specification. With respect to customers an inhibitor to component reuse is user reluctance to think in abstract terms, preferring more concrete representation.

Structure matching theory holds that there are attributes of structure that can be recognised and associated to gain insight from one particular problem into the possible solution of another, although on first inspection they might seem unrelated [22]. In this respect domain theory combines aspects of cognitive science and software engineering. A central tenet in cognitive science is that memory is organised in semantic networks of connected fact, called 'schema'. It is postulated that abstract schema are learned by people as they recognise and solve problems. Strategies for solving certain types of problem are built by an individual from some concrete source domain. The success of this strategy depends on the mapping of an analogue schema to a new target domain.

Sutcliffe presents an illuminating and persuasive argument. He draws on work on the theory of natural categories [23]. This is a powerful concept that underpins abstract thinking as articulated by the notion that a class is a mould on which instances are based, which has resonance with object-oriented programming. This is an effective approach for the categorisation of *things*, but not so effective for describing events and tasks [22].

Sutcliffe has drawn upon earlier work with Neil Maiden to re-present the ideas encapsulated by Object System Models (OSM) [19]. The soundness of OSM representation of real phenomena is drawn from the notion of ecological memory [24]. In DT, the similarities of systems that allow library loans are associated with those that allow car hire. Here the similarity is based on the transaction type. Library loans and car hire are defined as inheriting from the same basic schema. DT makes the link that a task model is equivalent to a script for achieving a system goal. The notion of cohesion is founded on the existence of a system to achieve a goal [25, 26]. A system goal represents an end-point in the life history of the transformation of an object [16]. It suggests an entity enjoys a life history that must achieve an end state although this may involve many intermediate steps. It may become possible to describe problems with a minimal set of objects and actions that satisfy all identified goals as the goal itself is a state.

By definition, an OSM cannot be de-composed; it is a primitive that has no sub-goals. OSM models have been constructed through a variety of means ranging through expert interview, investigation, and abstraction from problems taken from the SE literature. Further refinement has been undertaken via empirical testing.

One of the generic models defined in DT is that of the grounded domain. A grounded domain achieves a purpose at the application level such as solving 'hiring problems' represented in the examples of a library or video shop. Although these activities are clearly different, they share much in common. Extending this approach 'monitoring applications' include both air traffic control and hospital patient systems, which also are identified as having commonality; commonality of action rather than of structure. Another subclass is the 'Information system model' which manages reporting. A similar approach is described in knowledge engineering literature [27] where problem solving templates are introduced within the context of the construction of expert systems. Therefore, DT can characterise different business instances as inheriting from the same abstract problem schema.

It is postulated that each model will feature 'key objects'; an object essential to the transaction, managed by the application, which undergoes state change, from inception to the achievement of its final goal or stable state. The essence of the task in describing the satisfaction of goals is managing the state transitions in key objects. A goal can be described as a future desired state.

Object System Models (OSM) in Domain Theory

In Domain Theory, the concept of an OSM is used to define a common type of recurring business problem.

Below are listed the OSMs that have been identified.

- Object containment – uni-directional transfer of key objects from an owning structure to a client structure.
- Object inventory – orderly transfer of objects to client and replenishment.
- Accounting object transfer – like object containment, where a loan is like a hire but paid back with interest.
- Object returning
- Object hiring – control loan, hire or temporary transfer of ownership from org to a client

- Object servicing – repair. The key object is in some way improved.
- Object allocation – pre-cursor to some other state change i.e. reservation. Establish an association. Satisfying demands where resources are scarce. Constraint based matching. Optimise the abstraction of resources. Democratic sharing. First come first served etc.
- Object composition – aggregate key objects to synthesize new ones. Mfg raw materials – finished goods. ERP systems.
- Object decomposition – opposite of composition. Applications that 'dis-aggregate'. Disassembly, unpacking. How to remove. Order is important.
- Object construction – adds, deletes or modifies key objects. Whereas object composition only assembles objects. This is a large class that includes word processing, graphics applications and extends to decision support systems.
- Object logistics – move key objects from starting location to destination set. Key objects are trans-located within structure that represents the topology/geography.
- Messenger transfer – networks communicate key objects between structure objects.
- Object sensing – monitor the physical conditions or movement of an object in the real world and record their relative state with respect to a spatial or physical object structure.
- Spatial object sensing – divide up into 3D with air corridors, or 2D roads.
- Agent control – command and control apps. Controlling agent, controlled agents – military Object sensing, messaging, planning, analysing, modelling.
- Feedback and compliance. Consider needs to know the order has been carried out. Other problems. Unreliable sub-ordinates.
- Object simulation – represent model and behaviour to external agent.

This overview of basic OSMs is held to be a satisfied set because it has remained stable for a number of years. (Sutcliffe is willing to acknowledge there may be more, which have not been found as yet.) In effect, what is proposed is a reuse problem architecture, in the form of schemas, that has been verified over time against the experience of experts, and applied to case studies both in the literature and the field. Ultimately, Sutcliffe has high hopes for DT commenting that if it is to survive and grow as a respectable theory it must come to specify how every goal can be satisfied.

Modelling Languages and the “Method Wars”

Modelling is the act of representing applications before they are rendered in code. Building software models is important because the later a change is detected in the development lifecycle, the more it costs to put right [2]. A model is analogous to a blueprint used in the construction of a building. Models help verify that a development will meet the business requirements and the non-functional requirements (NFR). Functional requirements can be captured with use cases (*see*: Use Case Modelling on page 16). Conversely, NFRs can be captured with design patterns that describe ideal structures capable of delivering the desired level of scalability, robustness, security, and extendibility [28]. An application’s structure should ideally be defined to allow effective maintenance and possible extension.

In the 1980’s and 90’s, with the advent of object-oriented (OO) programming, came the need to develop new modelling languages for systems design based on this new technological paradigm. Many authors contributed methodologies for building object-oriented systems, all of which featured their own modelling language. Between 1989 and 1994, the number of methodologies went from less than 10 to over 50. A good review of the available methodologies available at the time can be found in [29].

Users found it difficult to gain complete satisfaction with any one modelling language. This period became known as the “method wars” because there were so many different approaches vying for dominance. By the mid 1990’s new iterations of methods began to incorporate each others techniques and a few prominent methods began to emerge. It became clear that much of the differences lay in notational style rather than content. For some practitioners it became clear that working together to create a unified approach to defining a modelling language was a sensible way forward. Three prominent methods of the time were the Booch method (Grady Booch), the Object Modeling Technique (OMT) by Jim Rumbaugh and Object Oriented Software Engineering (OOSE) by Ivor Jacobson. In 1994, Booch and Rumbaugh were working at Rational Corporation, a CASE (Computer-aided Software Engineering) tool vendor, where they began work to unify their methods. The objective was to eliminate, in a systematic manner, unnecessary and gratuitous differences of notation that confuse users. For Rational Corporation it was hoped this would bring stability to the CASE tool market place, allowing projects to settle on one mature modelling language

that would, in turn, help Rational to concentrate on adding useful product features to their software rather than simply variations of notational style.

Initially the work to define a single approach resulted in the Unified Method v. 0.8 of 1995, which although not entirely successful, suggested what might be possible. A year later Jacobson joined to collaborate on the less ambitious task of unifying their respective modelling languages (as opposed to the methods that underpinned the application of the languages) and the Unified Modeling Language (UML) v. 0.9 was published. These three collaborators, Booch, Rumbaugh and Jacobson became known as 'the three amigos'. Their work was submitted to the industry body, the Object Modeling Group (OMG), who were enthusiastic enough to issue a public RFP in light of the obvious advantages of working towards a unified approach. Other leading corporations now took note of the work being undertaken and submitted their own proposals; these include Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys and others. Duly, in January 1997, the amigos submitted their revised work to the OMG in the shape of UML version 1.0 which represented a wider consensus of industry opinion. (The current version of UML is version 1.5 while version 2.0 is awaited.) The success of this approach to defining a standard modelling notation is manifest. It has become the *de facto* industry standard, and there are moves afoot to have it adopted as a *de jure* standard under the auspices of the International Standards Organisation (ISO). The UML approach to modelling systems, even though it was launched into an environment of competition and disagreement, has proved itself a tremendous success and has been widely adopted. The Modelling Wars are over, and UML won.

A major strength of UML is that it is based on models that can be reliably transformed; each transformation progresses the project closer to implementation in hard, quantifiable code. The models that are produced have value to one or a variety of stakeholders depending on their technical expertise and the use to which the model will be put. Primarily consisting of object models, the original versions of UML did not offer any mechanism for representing user requirements. To be truly successful, any new language would have to incorporate notation that could represent not only the proposed solution, but also the application problem. The only method that offered a requirements capture mechanism was Jacobson's OOSE with its constituent use case diagrams; a notation for the representation of the problem (requirements) that could be grafted onto the existing notation.

Natural language

Prior to the definition of UML, requirements had typically been represented in natural language, an approach that had never been entirely satisfactory. A list of requirements expressed in sentences has no structure [30] and is often ambiguous. Sentences that begin with “The system shall...” or “The system will...” are inherently difficult to compare; made worse because they do not conform to any rules with respect to the level of abstraction at which they are expressed. This often leads to duplicate requirements being introduced that say much the same thing but use different words. Synonyms cause ambiguity when the lexicon of expression has not been defined. This means the relationship between what appear to be self-contained requirements cannot be made explicit.

Requirements are presumed to be independent and capable of implementation in isolation, although in practice, this is a dangerous assumption that often proves incorrect. Lists of requirements do not provide the customer or the developers with a cohesive view of the system under consideration and therefore are misleading. A requirements list provides no mechanism for check and balance with respect to what has been specified and what has been delivered. Natural language expression of requirements has failed customers, causing too many needs to be overlooked, being incapable of coping with requirements that change, and being implicated in requirements scope creep whereby projects balloon in scope and complexity over time [30]. Where the specification is written in natural language that forms part of a commercial contract, the degree of ambiguity can render the contract unenforceable.

Prototypes

Another technique in requirements gathering is the building of prototypes or ‘mock-ups’. However, prototypes are no substitute for structured requirements gathering [30]. It has been suggested that prototypes may be used to communicate requirements in procurement [31], although there is nothing in the literature to suggest this is common practice. There are two kinds of prototypes; graphical mock-ups that provide the user interface to the proposed system and functional prototypes that demonstrate actual behaviour. (Functional prototypes, by which algorithmic code is demonstrated, have no applicability to requirements gathering and are not further considered.) Graphical prototype development is via a succession of iterative releases that continue until the customer is satisfied all the requirements needed have

been captured. This approach may be appropriate where a supplier has already been appointed or where a system is built in-house, however, with a notable exception at the Inland Revenue [31]; graphical prototyping is not normally applicable during a procurement exercise, simply because they are costly to produce and unlikely to be undertaken without payment.

Later in the developmental process, Graphical User Interfaces (GUI) may have a role to play in verifying requirements have been understood. They can be useful in providing an illustration of the interface to the identified use cases [32]. For simple projects, those with limited complexity, prototypes may indeed be sufficient; however, these tend to be small-scale systems. It could be that in a small project the development may be undertaken by a single programmer without the need to share the work [33]. However, where a system is large enough that it will be built in teams, documentation that describes what needs to be built is essential. In this situation a prototype is useful, not as the primary mechanism of requirements gathering, but as a supporting tool. Prototypes have the drawback of encouraging users to focus on screen design details rather than the satisfaction of their *higher order* objectives, such as the goal or sub-goals the application must ultimately satisfy. One drawback of prototypes is that they tend to imply that something substantial has been built during requirements gathering, when in fact it has not [30].

Use Case Modelling

For many years, having recognised the limitations of natural language for requirements' expression, Ivor Jacobson had laboured on his ideas that would eventually come to be known as *use cases*. While working in Sweden for Ericsson, modelling telephone switches, those ideas came to fruition. He showed use cases to have significant benefits to customer and developer alike. Jacobson was so encouraged he left Ericsson and formed his own consulting company, Objectory, to apply more widely what had become known as Object Oriented Software Engineering (OOSE) [32]. Objectory came to have around 20 customers who continued to refine the notion of his new 'use case driven approach'. Objectory's customers included Swedish Defence, Ericsson Mobile, Ericsson Radar Electronics, and ABB, all of whom were able to demonstrate the approach's merit.

The use case approach is iterative; introducing complexity gradually [32]. This iterative process is in accordance with the way the task is actually performed by practitioners. It has been shown that the

customer is in no position to fully describe their requirements. Requirements become more clear over time [30]. As the process continues more detail is added (perhaps through de-composition) but, overall, the highest level use cases remain stable. Eventually the analyst has enough information to transform the use case model into sequence diagrams. This transformation from one model to another *type* of model is a key factor in the attractiveness of UML. The transformation of models indicates the project is moving closer to implementation, where instructions as to what should be coded are sufficiently detailed. Sequence diagrams are well suited to representing communication between objects by specifying their public interfaces and the passage of messages. Sequence diagrams are a more concrete realisation of the structure necessary to deliver the functionality described by use cases. They are the link between the problem representation and the first artefact of a proposed solution. Sequence diagrams were also an innovation introduced in OOSE, that Jacobson modified from electronic modelling that proved a natural compliment to use case modelling.

Use cases can be thought of as an initial model of the problem under consideration that sits at the centre of the 'developmental wheel'. They allow requirements to be represented in a graphical form, that can be transformed into solution specific models from which code can be written. They can act in a project management capacity to keep track of progress throughout the developmental lifecycle. They can be included in commercial contracts, and can inform user documentation. Use cases can be employed as test case scripts [34]. Use cases can be the basis of measuring progress, the basis of parcelling up work for different teams working in parallel, and the basis of traceability. By any measure, use cases are a versatile and flexible mechanism that offers a unique contribution to software engineering that features a solid route from project inception to implementation. Use cases have proved to be a notation that is understood by business and by technical staff [34], thereby providing a joint language of communication that suffers less from ambiguity.

RUP

Where in the past the notions of methodology and modelling language had been inextricably linked, with UML they were de-coupled. The approach of the OMG is to advocate the use of UML without imposing a methodology that defines how the process of building software should be accomplished. In effect, what is available is a notation for expressing how software is built, and a complimentary methodology that

describes when the models should be built and their dependencies. The Rational Corporation, the driving force behind UML, has defined a methodology known as the Rational Unified Process (RUP) that can be used to manage the totality of the process, although use of RUP is not a prerequisite of employing UML. The models produced with UML are the artefacts of the RUP method, but could equally be the artefacts of another method, such as Prince 2 [35], should the development team wish. This separation of method from language has done much to encourage the uptake of UML.

The Business Case

The purposes to which use cases are put are varied, being both explicit and consequential. However, the importance of use cases goes beyond an engineering methodology; use cases also have a commercial dimension [36]. They can be used as the basis of a RFP in commercial tendering with the expectation their use will result in a specification that is more succinct and less ambiguous [37, 38]. They can be used in commercial off-the-shelf (COTS) package selection to compare requirements with available features. This can aid understanding in deciding the fitness for purpose of the alternatives and the degree of required customisation that may be necessary. Kulak and Guiney see use cases having a major impact on commercial decision making and in contract negotiations [30]. Failure to capture requirements comprehensively and unambiguously makes a commercial contract very hard to enforce; the parties are left to squabble about what exactly they *meant* when they originally described what was wanted [39, 40].

The employment of use cases rather than requirements lists, as the basis of commercial contracts, appears to have merit, because use cases can more easily be argued to represent behaviour that can be tested. Use cases can be rated on the basis of priority to different stakeholders, thereby demonstrating progress to those whose continued support is necessary to the continuation of the project [36].

The amount of time and effort needed to build a system is strongly related to the amount of functionality the system will exhibit. The more complex the system, the more time (and money) it will take to build. As a notation, use cases represent function (or *goal satisfaction*). It is possible to employ use cases as the input to an effort estimation algorithm [41-43], although there remain issues to be resolved before this can be done routinely and with reliable utility.

The Engineering Case

The primary objective of employing use cases is to adopt a requirements capture process that is easy to understand and to formulate. No system should proceed into design until the requirements are completely understood, if for no other reason than fear of building a system that does the wrong thing. Although originally developed while working on telecommunications systems, use cases are also well suited to the representation of transaction processing systems. To aid use case modelling the notation has been kept simple. It is intended that the picture produced should be easily ‘surveyable’ and changeable [32]. Use cases are the focal point around which discussion with users takes place to discover their requirements. An initial model can be produced quickly and easily at the beginning of the process to determine whether users are satisfied with the representation. It highlights unclear points in the requirements specification and allows the right questions to be asked to eliminate ambiguity and develop an increased understanding of the problem. Use cases are the mechanism by which the supplier can assure themselves they are developing a system that will satisfy the end-users' real problems.

Use cases can be employed throughout the lifecycle to benefit different stakeholders as illustrated in Table 1.3.2. One approach to the construction of use cases is to identify those that represent the greatest degree of technological risk and to build these first [36].

Table 1.3.2: Ways in which use cases are employed.

Business Applications	Engineering Applications
As the basis of RFPs	As the input to the creation of a domain (entity) model
In contract negotiations and agreement	As the input to the creation of a robustness model
As the basis of effort estimation	As input to other diagrams in the UML engineering process (domain models, sequence and activity diagrams)
As the unit for the measurement of productivity	As the basis of test cases
As the basis of the measurement of progress	As the mechanism to assign work for parallel development
As the basis of prioritisation	

What is a use case?

Many definitions have been postulated for the definition of a use case, yet still the approach has been criticised for a lack of formality in definition [44]. Originally Jacobson described a use case as “a specific way of using the system by using some part of the functionality. [A use case] constitutes a complete course of inter-action that takes place between an actor and the system” [32]. Later, in the UML, the subject of a suitable definition was expanded when it was suggested that “a use case is the specification of sequences of actions, including variant sequences and error sequences that a system, sub-system, or class can perform by interacting with outside actors [45]. Philip Kruchten provides a short elegant definition when he says that “a use case yields an observable result of value to a particular actor [46]. Martin Fowler sees a use case as “a typical interaction between a user and a computer system [that] captures some user-visible function [and] achieves a discrete goal for the user” [10]. Constantine and Lockwood provide a definition that is more complete at the expense of being less easily digestible. They say that a use case is a “single discrete, meaningful, well-defined task of interest to an external user in some role in relation to the system, comprising the user’s intentions and system responsibilities in the course of accomplishing that task, described in abstract, technology free implementation-independent terms using the language of the application domain and of users in role” [47].

A use case is a description of a sequence of actions that a system performs to yield an observable result to an actor. They do tangible work [calculate result, generate new object, change object state...] [48]. A use case has a basic (normal) flow of events, a set of alternatives, and a set of exceptions. Use cases capture functional requirements; they do not capture business rules, performance or complex algorithms [49].

A use case is a representation of a user goal to be satisfied. A system can be considered a collection of use cases together represented in a use case model. The use case model is a picture intended to be easily ‘surveyable’ and changeable by customers and developers alike [32]. A use case model (UCM) has actors, task ovals, associations and a system boundary. UCMs start simple and become more complex over time [34]. Each use case has two parts; a graphical representation and a textual representation. The text part adds detail to the graphical representation. It is convenient to consider the graphical component of a use case as a kind of *table of contents* that directs the reader to the accompanying text.

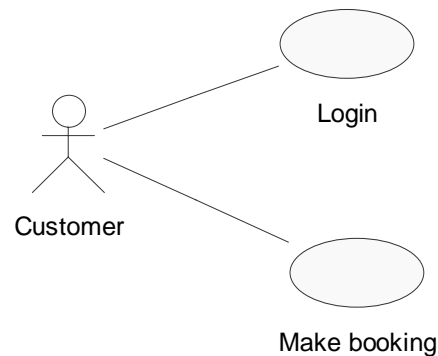


Figure 1.3.1: A simple use case model, featuring an actor, two task ovals, with associations.

In Figure 1.3.1, the most basic form of a use case is represented graphically by a named task oval that represents a user goal. This diagram illustrates a customer's ability to *Login* and to *Make booking*. As more detail becomes available, it is added to the use case. Over time, the use case becomes primarily a textual construct that describes the system behaviour in semi-formalised natural language.

Actors

Use cases are triggered by actors. Jacobson defines an actor as a construct that represents a role a user may play [32]. An actor who triggers a use case is a human user who plays a well-defined role. Human actors are also known as primary actors. There are, in addition, secondary actors that represent other software systems with which communication must be established. Lastly, there are secondary actors that represent encapsulated behaviour; consider the concept of a *clock* actor that automatically triggers use cases, or an *email engine* that offers common behaviour to many individual use cases to communicate. To Jacobson, actors are simple to discover, coming from an analysis of customers, partners, suppliers, authorities, and subsidiaries [36]. Actors are considered outside the system boundary, although primary actors may come to have an internal representation even if it is on the trivial level of permissions, such as in the case of a notional *customer* who would need to have permission over their own account. It is useful to distinguish the ultimate primary actor; the actor who 'really cares' about the successful completion of a use case, rather than a proxy actor who is acting in their stead [50]. Thus, a clerk can be regarded as a proxy actor on behalf

of a customer. This leads to the development of primary actor hierarchies which are related via specialisation.

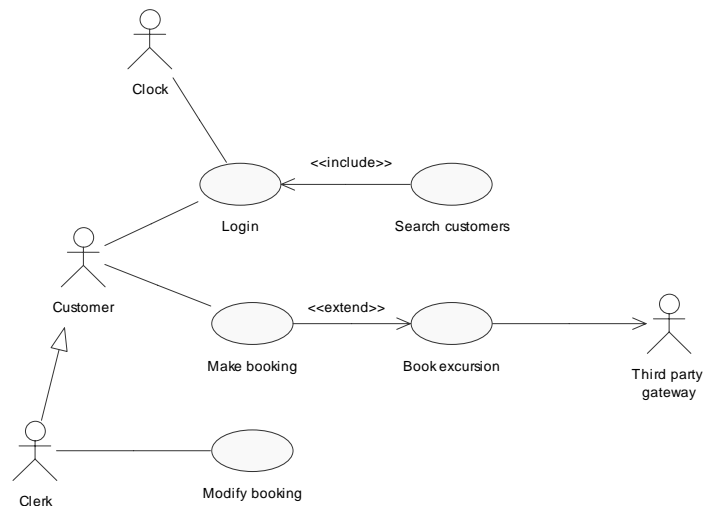


Figure 1.3.2: Actor types as part of a hierarchy. The Clerk and Customer are both primary (human) actors. The 'Third party gateway' is a legacy system that accepts excursion bookings. The Clock is an automated convenience that logs users off the system when they have been inactive for a period of time.

Actors may be realised by devices, other systems, or by human beings; all three of which are illustrated in Figure 1.3.2. Actors are connected to use cases only via association, meaning they communicate by both sending and receiving information. Primary actors have a goal they wish to see fulfilled. Secondary actors assist in realising the goals of primary actors. A use case task oval is associated with a text description. An actor triggers a use case instance, termed a scenario, which performs a transaction on the actor's behalf. Cockburn considers that the basic course of action can be thought of as the 'archetype scenario' [50].

Use Case Model

A collection of use cases is represented in a diagram appropriately named the 'use case model' (UCM); this model represents the scope of the project [34]. A use case model is a collection of all graphical use cases. It shows the relationship between actors and between tasks. It also defines an actor's role, (permissions) and the inter-relationships between actors. The model allows similar use cases to be grouped together and be managed in UML packages (folders). These packages are a convenient device for parcelling up sets of use

cases that may be assigned to individuals or teams to facilitate parallel work [34] while at the same time minimising the risk of duplication of effort.

Domain Objects

The use case model is extremely useful in identifying *domain* objects [32]. Jacobson defines three types of domain object, (roughly analogous to those identified in the model-view-controller design pattern [51]); *interface*, *control* and *entity*. Entity objects are logical representations of database tables without the complication of building a fully normalised data model. Interface objects represent the user interface, and control objects mediate between interface and entity objects.

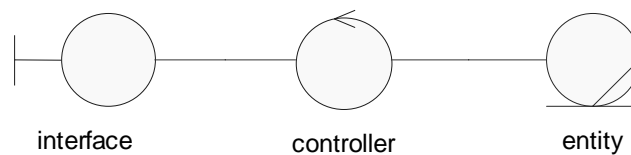


Figure 1.3.3: A domain model featuring stereotypes of the interface, controller and entity types.

With reference to Figure 1.3.3, the functionality of a use case is partitioned over these three logical types, or 'domain entities' [32]. An entity object represents data held over time that survives the execution of any particular use case. An interface object provides the trigger to a use case. A control object operates on several different entity objects; it may perform some computation and returns the result to an interface object.

Use case text

A use case has two parts; graphical and textual. The textual description defines the basic success path through the use case, along with any alternate paths and exceptional paths. There are different templates available for the representation of the textual component of a use case; they will all normally include details of extension points, triggers, preconditions and post-conditions. Not all information is available at the beginning of the lifecycle of a use case. More detail (precision) is added to the use case as a project proceeds [50]. It has been shown that the adoption of templates makes for use case text that is more easily understandable [52].

For the sake of clarity, some definitions are offered. The textual components of a use case can be termed collectively 'text cases'. There are three kinds of text case; exactly one *normal* case, plus *alternative* cases and *exceptional* cases. These text cases are written in abstract language; they do not feature named instances of actors or example values. A text case is phrased like a class of behaviour such as "the customer enters their secret code number", rather than "Dan enters the digits 2198". In the majority of instances in which the use case is triggered, it is the normal case that executes to a successful conclusion. Alternative cases start with the normal case execution, but then jump to the alternative behaviour based on some condition being true (the user enters the wrong identification number) that causes some alternative behaviour to be triggered (the machine displays a message asking for the security number to be re-entered). Exceptional cases are reserved for behaviour that describes a failure in the system (the network is unavailable).

Scenarios are based on the use case narrative defined in the text cases. A scenario is an instance example of a use case being executed by a named individual performing a business function in a prototypical situation substituting actual values for named variables. Optionally, a scenario can be written for each pathway through a use case, although a scenario for the normal case will often suffice. Scenarios put an abstract description into user language through examples in the form of user stories. Scenarios are typically extended narratives forming a plausible vignette or storyline.

Unlike text cases, a scenario is more like an example of a story that might be told informally between two people describing the behaviour of a system. Scenarios place example individuals and example values into the description found in the text case (Dan enters the digits '2198', the machine verifies this is the correct number for Dan's account, the machine displays the options to withdraw money or to check the account's balance, etc.) Customers are liable to prefer scenarios to text cases, finding them easier to read and hence more accessible [47].

A transaction is a term taken from the study of database design to refer to an action that is either completely performed or not performed at all. In the parlance of use cases, a transaction could be the execution of the normal case, or of an alternative that starts with the normal case and then terminates by executing the steps

defined in an alternative or exception case. Some authors on the subject of use cases employ the word ‘transaction’, which means a complete valid path through the available text cases [41].

Table 1.3.3: Typical Use Case Template to help structure the use case text. Many authors have proposed alternate templates [30, 50, 53]. A good discussion of template styles is given in [52]. The contents of the templates varies somewhat, but essentially include the headings represented in this table.

Use Case ID:	
Use Case Name:	
Actors:	
Description:	
Trigger:	
Scope:	
Summary:	
Preconditions:	
Postconditions:	
Normal case (basic flow of events):	
Alternative cases:	
Exception cases:	
Extension points:	

In table 1.3.3, the use case identifier *Use Case ID* can be used to assign each use case a unique numerical identifier. *Actors* who participate in the use case can be listed. A brief description of the use case can set its purpose into the wider system context. *Preconditions* list any activity that must take place, or any conditions that must be true, before the use case can be started. An example would include the fact that previously the user’s identity had been authenticated. A *postcondition* describes the state of the system at the conclusion of the use case execution. An example of a postcondition is that the price of an item in the database has been updated with a new value. *Normal case* is a text case also referred to as the *base case*. An *alternative case* is behaviour that may occur when the base case fails, but where the outcome is not an error. *Exception cases* describe possible error conditions and subsequent behaviour that results. *Extension points* can be used to indicate the places in the normal case where the use case calls another extending use case.

Narrative style

Working from a template provides some level of consistency across different use case instances [52]. Although templates may specify what should be included, they do not define how the text cases should be written. This may lead to substantial differences between use case text written by the same author, exacerbated when many authors are involved. Templates are of no help with respect to narrative style, as each case, normal, alternatives, and exceptions, must still be represented with natural language. Similar problems to those already identified with natural language persist with use case expression at this level. Knowing what to include in a text case is not defined by templates. Without guidance, text cases tend to intermingle analysis and design, business rules, design objectives, and interface descriptions “with gratuitous asides thrown in to cover all bases” [47].

Common narrative styles have included a continuous free form narrative description of the flow of events. A variation on free text description is that of numbered sequence text [46] where discrete steps are defined and responsibility assigned to either the actor or the system. Both approaches have been criticised as being too wordy, of introducing verbiage that sheds little light, but substantial noise. Authors have slavishly included an opening and concluding line in each case, something in the form of “The use case begins/ends” has been labelled a redundancy [47]. Instead, Constantine and Lockwood, drawing on the earlier work of McMenamin and Palmer [54], who first introduced the concept of the essential text form believe this structure can be omitted without losing any meaning. The essential form is more akin to pseudo-code but without specialist programming constructions such as ‘until_done repeat’ commands that look inappropriate to the lay reader. The advice is that text should not look like code but that it should be concise and minimalist.

Applying the essential form to case text authoring the objective is to write abstract, generalised, technology free, descriptions of the essence of the problem. Other authors have identified that there is a need to prioritise abstraction in case text writing providing momentum towards the adoption of an essential form [50, 55, 56]; doing so would allow case text to be more easily compared. To accomplish this move to an essential form, text should be written in a terse style, omitting all implementation details. Over time, through iteration, each text case may acquire more detail, thereby featuring expression of a more florid

style. An essential use case can be suitably vague, as it omits details of the user interface (UI) design. The objective is to keep to user intentions and system responsibilities that lead to the accomplishment of a goal, defined as a stable end-state. In this sense, goals are destinations and the intention of the user represents their journey to the satisfaction of the goal. Two examples are presented below from [47] that illustrates the difference in the representation of the same use case.

Example 1 – unstructured style:

The use case begins when the customer goes to the Customer Log-On page. There, the customer types in his or her name and customer ID and submits it. The system then displays the Tech Support home page with a list of Problem Categories. The customer clicks on Installation Help within the list, and the system supplies the Incident Report Form. The customer completes and submits the form, and the system presents a suggested resolution.

Example 2 – essential style:

User Intentions	System Responsibilities
identify self as customer	
	present help options
select help option	
	request description
describe problem	
	offer possible solutions

Not everyone agrees that use case text written in a terse style is a superior form [57], but when coupled with the ability to later provide elaboration as required, it appears to be the best strategy. Biddle says that essential use cases are more likely to be reusable [58].

There are other problems with case text definition such as describing behaviour as though it were strictly ordered when, in fact, it is not [44]. User behaviour may be described as though it were a step by step process when in fact it is common that different tasks can be performed in any order without affecting the successful completion of the task. Some analysts use conditional and iteration expressions to allow the expression of a series of sub-tasks that are not governed by order.

In any order (specify part no; specify quantity ordered)

Case text can be viewed on a continuum of abstraction where the essential form is entirely abstract, unstructured natural language case text is moderately abstract, and scenarios (user stories) are grounded in the user experience and so are not at all abstract (they are concrete). End users normally find scenarios the easiest to understand as they are written in every day language. The essential approach favours a fine-grained decomposition of task with partition into collections of simple use cases that are inter-related by inclusion, specialisation and extension over time [47].

Associations

In addition to the elements already described in a basic graphical use case (Figure 1.3.1) use cases themselves may also be associated through the <<include>> and <<extend>> stereotypes (Figure 1.3.4). This association between use cases is intended to be useful as a mechanism for reuse. Jacobson has recently postulated that such ‘internal’ use cases (not directly triggered by an actor) should not be considered as use cases in their own right, but rather as use case *fragments* [59]. In Figure 1.3.4, the <<include>> and <<extend>> association are illustrated.

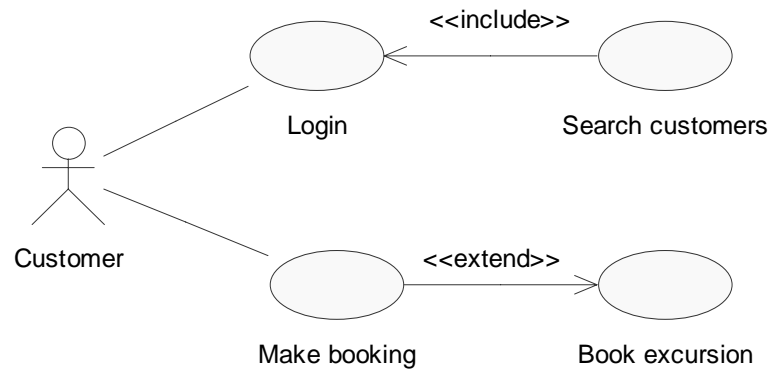


Figure 1.3.4: A simple use case model with an <<extend>> and an <<include>> stereotype.

Originally, use cases could be associated in two ways, either through the <<use>> or <<extend>> stereotypes [32]. Jacobson introduced these associations during his time at Objectory. In UML 1.3 the <<use>> tag was replaced with the <<include>> [10] tag (probably to limit the word ‘use’ which had already acquired many connotations). Although Jacobson saw these stereotyped associations as being straightforward, they have since been the topic of considerable confusion and ambiguity. Jacobson believed

the system modeller should associate two use cases with an <<extend>> stereotype when the extending use case represents additional behaviour. He suggests the extending use case should make sense on its own, that it represents new functionality being added. He believed this was a mechanism that would allow a system to grow (be extended) over time and where what was added would not require changes to what already existed [32, 59].

In considering the <<include>> association, Jacobson saw an included use case as making no sense on its own; that it must be *called* by the containing use case in order that it should perform something meaningful. It is sensible to extract *included* behaviour because it represents something common that can be reused [32, 59]. Cockburn agrees, seeing <<include>> as employed when there exists a ‘chunk’ of behaviour that is similar across more than one use case and rather than defining behaviour over and over again it can be associated to a containing use case with an <<include>> [50].

On the face of it, these associations do not appear to represent concepts that are difficult to apply, but there is evidence to suggest that they are not well understood in practice [60]. Cockburn says that extension is an option when one use case is similar to another but it does a bit more. This sounds suspiciously similar to the test employed in specialisation where the modeller has the choice to either break the new specialised use case out of the containing use case or to deal with the specialised behaviour as an alternative within the same use case. He gives no guidance as to when to use one approach and when the other. By this definition a specialised use case may be said to describe *alternate* behaviour rendering the decision unclear. The trouble comes when a modeller attempts to put the guidelines to the test; questions are uncovered to which no adequate answers are readily available. There is a problem in deciding how *alternative* behaviour should be modelled. Given that use cases are associated through <<include>> and <<extend>> associations they become use case *fragments*, rather than *first order* use cases in their own right. However, examples have been presented in this work where associated use cases are both fragments and of the first order (*see*: Figure 1.3.3 and Figure 1.3.9). The rationale for employing an <<extend>> is easier to define when reference is made to implementation. It makes no obvious sense that an excursion can be booked unless it refers to a particular holiday. For example, booking an excursion is in addition to the normal case of booking a holiday; it is clearly additional behaviour. In execution the normal case executes through *Book Holiday*, and then the normal case may run in *Book Excursion*.

Light can be shed on this ambiguity by examining the differences between the <<include>> and <<extend>> stereotypes. They are not inverse associations. Where <<extend>> is taken to represent optional behaviour, the assumption that <<include>> represents mandatory behaviour is attractive, but incomplete. To reconcile the positions it is necessary to return to first principles. A use case has only one normal case where the user goal is *fully* satisfied. Alternatives are not equal to the normal case, as they represent another outcome which is *an* outcome (which is certainly not an error), but is not as successful as that of the normal case (the 'happy path' [61]). Alternate cases are therefore execution paths where the user goal is only partially satisfied.

Behaviour associated with an 'external' use case through <<extend>> takes the normal case behaviour further, thereby combining two normal cases. In this sense it is optional behaviour, but it is not optional to the execution of the external normal case; it is in addition.

Behaviour associated with an 'external' use case through <<include>> is mandatory to the execution of the normal case, and perhaps also to the alternate cases. Included behaviour is likely to become apparent over time, when the text cases are written, and is unlikely to be identified during *façade* modelling (a *façade* use case is one that features only the graphical component). The modeller/author will notice that they are continually writing the same text; this becomes wearisome and so the decision is made to break the repetitive behaviour out into a fragment that can be associated through an <<include>> stereotype, if for no other reason than because it is more efficient.

The subject of associations between use cases is full of ambiguity. This leads to confusion in model-making. Two modellers confronted with the same problem must inevitably create two different models. This leads to the problem of evaluating which model is superior and why. The subject of associations between use cases has been largely accepted as a non-contentious subject by many but there is evidence the subject is not clear [57]. One author goes so far as to counsel that the subject of stereotyped associations receives too much attention, that it is a distraction that can wait until an advanced state in the modelling [34], and that it is a fundamentally boring argument!

Glinz poses some of the most cogent criticisms of use case modelling [62]. Many of the rules that govern it are too restrictive and routinely broken. The rules [63] were intended to encourage practitioners to model

with use cases in a defined manner taken from an agreed specification written in wide consultation. Still, the restrictions are too onerous; compromising the richness of expression needed. This is a specific criticism of the omission of inheritance and aggregation from the association toolbox.

Use Case Discovery

Knowing the components of a use case does not address the process by which use cases are discovered. To enter into the discovery of use cases it is helpful to consider them as being underpinned by a semi-formalised language based on grammatical constructs.

In considering the naming of a graphical use case, the task oval component must be uniquely identified. The form of the name should include an active verb, in the present tense employing the active voice that names a recognisable user goal [57]. Jacobson favours the employment of singular nouns and verbs in the infinitive [32]. There is a balance to be achieved in the selection of concrete vs. generalised verbs. Alongside the chosen verb goes a suitable noun or noun phrase (e.g. Book excursion). A use case name should be an instance of identifiable and acceptably atomic behaviour of the system. In practice, use case names are short active verb phrases naming some behaviour found in the vocabulary of the system being modelled. To name a use case, one should employ a simple verb/noun construction [32]. The objective is to avoid ambiguity in the naming of task ovals, successfully achieved when the name is self-evident to the majority of users [47].

The user goal may be described at different levels of specialisation/generalisation. Thus, some use cases are abstract, incapable of being instantiated themselves [32], but providing a hierarchical structure for efficient use case management.

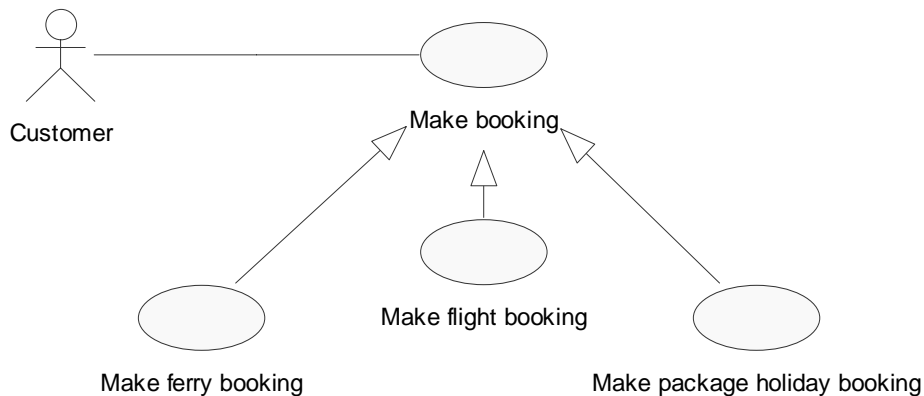


Figure 1.3.5: A use case structure where the *specialisation* association has been employed to add more information about the use case relationships.

In figure 1.3.5, the *Make booking* use case is abstract (cannot be instantiated) but it acts as a convenient placeholder to illustrate that there is some commonality between the different types of possible bookings. The different booking types are represented by three *specialised* use cases with significant differences subject to different business rules in the detail of *Make ferry booking*, *Make flight booking* and *Make package holiday booking* over its parent *Make booking*. The customer is understood to trigger the specialisations rather than the parent use case. This is a succinct approach to diagram construction that would otherwise require the actor to be connected to each specialised use case, inevitably cluttering the diagram.

All systems have some generic functionality that must be considered such as the ability to *log on* in order to gain access to specific permissions. Other generic functionality includes provision for security, audits, backup, remote access and reporting requirements [30]. However, the process of unique use case discovery starts with the original requirements statement [32]; a document normally written by the customer (and subject to variety in the quality of expression). Presuming this document is of adequate quality, the first use case model is developed from the requirements statement to reflect a view of the functionality as seen from the perspective of actors. A common recommendation for use case discovery is to traverse the actor set and consider the goals of each in turn [50], beginning with the customer (or consumer) actor [36].

Another, more mechanistic approach, is to identify entities (certain nouns) from the text, to create use cases to manage the entities and to then assign the use cases to the actors [34]. Following the common process for discovering objects, the nouns in the use case name make a solid basis on which to derive the identities of persistent entities [7]. This does not discover all the objects in the system, but it does uncover the majority of the persistent objects (those represented in a database) and then represented in a logical domain model as illustrated in Figure 1.3.6.

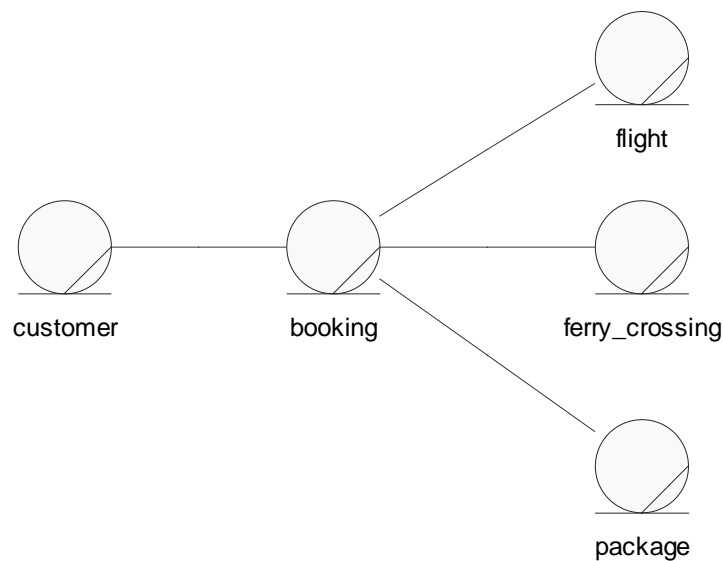


Figure 1.3.6: A simple logical domain model that is useful in discovering use cases such as Create customer/booking/ferry crossing etc.

CRUD functionality

Most systems store and retrieve information (derived from stored data), a large class of which are transaction processing applications [17, 18]. Different actors want to interact with the same data; some of whom have the access rights to make changes, others who may only view data but not make changes. All data must be subject to the standard considerations that apply in relational database design, namely functionality to create, retrieve, update or delete (referred to as CRUD functionality) although there is no agreement on the degree of importance of use cases that perform this kind of management over data [30, 50, 61].

Cockburn provides a neat encapsulation of the argument around CRUD use cases when he considers the management of an imaginary entity called a 'frizzle' [50]. He says, "So far there is no consensus on how to organise all those little use cases of the sort *Create a Frizzle*, *Retrieve a Frizzle*, *Update a Frizzle*, *Delete a Frizzle*. These are known as CRUD use cases, from the Create, Retrieve, Update, and Delete operations on databases. The question is, are they all part of one bigger use case, *Manage Frizzles*, or are they separate?" Cockburn specifically discusses this issue of CRUD representations, coming down on the side of 'manage' to represent the functions of create, retrieve, update and delete over a persistent entity. Other authors believe it is best to keep them separate to better aid identification of which actors have what permissions. Accepting that create, retrieve, update, delete should be kept separate has the disadvantage of multiplying the number of use cases that need to be tracked, and does nothing to improve the clarity of the resulting diagram. Given that *manage* use cases exist, the problem arises of where and how they fit into a use case model [30, 49, 50, 61, 64]. Although making available the functionality to *manage* data entities is necessary, this was not Jacobson's vision of how it should be done. Therefore there is a tension in the articulation of use cases, between the representation of user goals at the *appropriate* level and at a *pragmatic* level. CRUD use cases can be understood as pragmatic. More discussion is necessary before it is possible to approach use case definition at an *appropriate* level. For the time being, it is enough to recognise that use cases must be identified that are both.

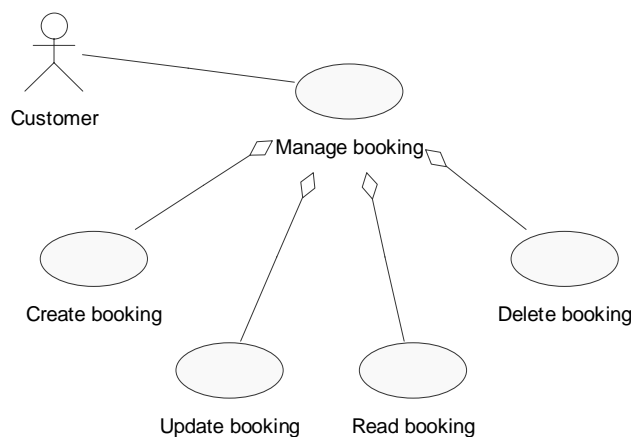


Figure 1.3.7: A set of CRUD use cases over the entity *booking*. The *children* use cases are shown related to the *parent* use case via the aggregation relationship.

Employing the *Manage* tag to represent CRUD functions can make the diagram less cluttered as in Figure 1.3.7. Aggregation has been employed to associate the sub-use cases to the container *Manage booking*. (Aggregation is not permissible in the UML standard, but is employed in Figure 1.3.7 to make the point that ‘Manage booking’ is comprised of other use cases. Sometimes it is necessary to show all the constituent use cases contained in *Manage* to deal with actor permissions. This diagram implies the Customer has total control over a booking, a situation unlikely to exist in practice (no company would want a customer to be able to make unrestricted changes to an existing booking).

CRUD use cases cause controversy. Kulak and Guiney [30] state it is a mistake to begin use case modelling with *manage* use cases believing it to be a sign of over-engineering. However, accepting *manage* use cases exist as a first principle is fundamental to Biddle [64] simply for the sake of completeness. It is not essential to reuse the standard language of relational databases to describe maintenance functionality over an entity. If create, retrieve, update and delete do not capture the functionality required, the specifier should change the names to better reflect the domain. For instance there may be more than one way to create an entity, either by making a new one, or by copying an existing one. Providing utility functionality over entities should not be done slavishly, without first considering the nature of the discovered entity, however, it is likely to be an area of functional specification that will lend itself to a degree of automation [58]. The full range of *manage* use cases are not required over all the identified domain objects, such as the example where a ‘time period’ is modelled as a first order entity (parent table) rather than as an attribute of some containing table. Thus, the essential task of ensuring the domain model is a minimal representation of the entities needed to manage the business rather than a fully normalised database that includes parent – child relations which are inappropriate at this early stage of modelling.

It is important in the beginning to understand how to begin collecting use cases, how much detail to collect, the level of abstraction to employ, in order to determine when it is safe to move on.

Abstraction

The dictionary definitions of abstraction are intended for a broader audience than that of object modellers and systems analysts, yet it is helpful to return briefly to first principles before exploring the wider issues of abstraction in, specifically, use case expression. Abstraction is the ‘act of leaving out of consideration one or more properties of a complex object so as to attend to others’, according to the definition offered by Webster’s Revised Unabridged Dictionary [65]. From WordNet © 1.6 [66] abstraction is ‘a concept or idea not associated with any specific instance’ and ‘the process of formulating general concepts by abstracting common properties of instances.’ To the Free On-line Dictionary of Computing [67], abstraction is defined as ‘generalisation; ignoring or hiding details to capture some kind of commonality between different instances.’

Representing use cases at the same level of functional abstraction is difficult [34]. Use case modelling is full of different concepts that are abstractions to one extent or another. Within the world of object modelling, abstraction is generally understood as inheritance (generalisation/specialisation) and aggregation (whole-part structures).

A use case is primarily a mechanism for representing a goal abstracted to a level that makes the most sense to the user (or other stakeholder). One challenge is to define use cases that are comprehensible at the user level and useful at the developmental level. The question arises, how is it possible for the same construct to satisfy both? One way is to imagine use cases arranged in a hierarchical structure where they can be decomposed into sub-use cases (aggregation). Use cases can also be thought of from the perspective of inheritance. Although use cases are not defined as being able to apply either inheritance or aggregation, there are advantages of doing so.

Concerning inheritance, a hierarchy may be defined according to the strictures of an actor satisfying a goal expressed in the language of the domain. This allows representations such as [make booking, make *specific kind* of booking]. Aggregation, in the form of a single user goal use case being sub-divided into several sub-goal use cases is equally a helpful concept to be able to apply when modelling. A sub-goal use case may be of no interest to a user; interesting only to the development team who are charged with implementation.

There are other applications of abstraction in use case modelling apart from those that associate use cases.

The concept of *actor* as role is an abstraction away from the specific concerns of a named user instance (e.g. Gary), that allows the modeller to concentrate on the characteristics many individual users share, such as their job description as comprised of the tasks they must accomplish (e.g. 'Clerk' *see*: Figure 1.3.3).

The idea of abstraction informs the writing of text cases with respect to scenarios. The text cases define the collection of possible outcomes that may arise through the invocation of a particular use case. A scenario, however, is not abstract, it contextualises a text case by referring to a named user carrying out an identified task with a specific outcome and referring to actual variable values.

Abstraction is the omission of detail. In this sense, use cases are abstractions, as they begin with little detail and gain more as time passes. Kulak and Guiney describe this process of moving through a continuum of increasing detail in their four 'F' model of *façade*, *focused*, *filled* and *finished* stages of a use case [30]. In each stage, a use case gains detail. In the beginning, the *façade* stage use case is solely a graphical construct (all use cases presented in diagrams thus far have been *façades*). In the *finished* stage, a use case contains everything, including the completed text template.

The *façade* iteration includes the creation of a model, taken from the original problem statement. The problem statement is constructed from existing documentation and intellectual capital. It will have been used to canvas the sponsors' viewpoint and for the identification of users, stakeholders and customers. The problem statement is used to find actors and suggest domain entities. The tasks the identified actors want to fulfil are identified by understanding their job function [30]. The purpose of the *façade* iteration is to create graphical *placeholders* or use case names and short descriptions characterised as including minimal detail.

The abstraction of use cases based on detail is not the only way it has an affect. There is also the abstraction based on the project stakeholder or project goal. Consider the question, "how many use cases does the typical specification contain?" Jacobson believes a system will have between 10 and 20 use cases [32, 59]. John Smith at Rational Software defines the ideal number to be between 10 and 50 [68]. Smith

states that a large number (100+), indicates a ‘lapse into functional decomposition’²; a state that is to be avoided [50]. However, Cantor in [69] makes clear that different processes are referred to by the term ‘functional decomposition’ and that no universal definition is accepted. The numerical discrepancy in the identified range of the ideal number of use cases in a specification, differs because use cases can be expressed at different levels of goal abstraction. Where a user goal use case is modelled with inheritance or aggregation, the effect will be to increase the overall number of use cases that appear on the model.

When counting use cases in a model, Smith employs the ‘external use case’ test. This is defined as a use case that has a direct association with an actor. Any use case that does not have a direct actor association is an internal use case (or fragment) that should not be counted as a *first order* use case. Unfortunately this convenient convention is exploded when one actor’s external use case is another actor’s internal use case. This is illustrated in Figure 1.3.8 and Figure 1.3.9.

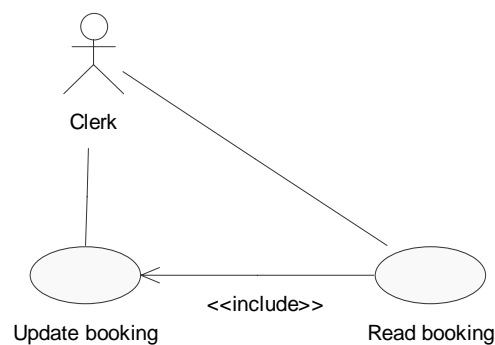


Figure 1.3.8: A sub-set of basic functionality triggered by the Clerk.

2 Functional decomposition: There is no official definition of functional decomposition. The term is used to describe several activities including adding more detail to a general requirement, organizing requirements into packages, and determining the organization of subsystems. There is likely to be little harm in the first two, which are ways to better manage requirements. However using functional decomposition to derive a system architecture is a bad strategy that may jeopardize the project.

In Figure 1.3.8, the Clerk may either Update booking, which includes ‘reading’ over the set of all bookings (searching for a booking to find the correct one), or may have, as an ultimate goal in its own right, to find a booking so that some detail may be confirmed to a waiting customer. Therefore, Read booking is both an external and an internal use case.

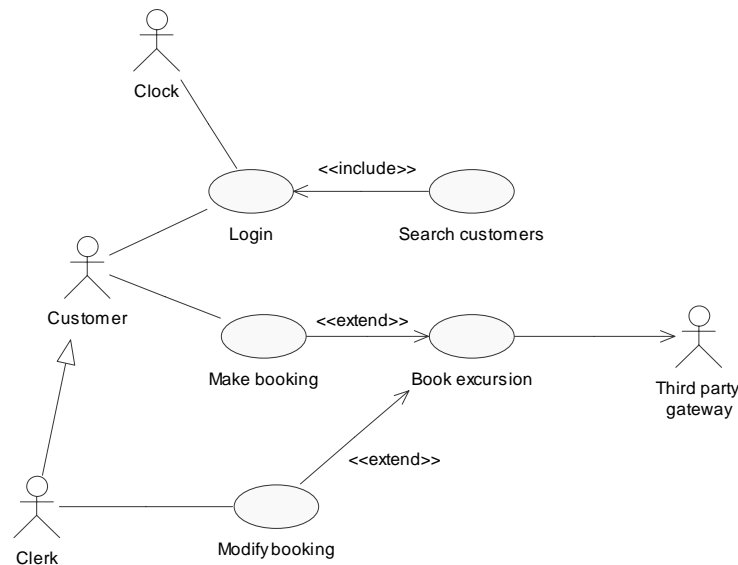


Figure 1.3.9: An extract of the total functionality delivered by the booking system.

Figure 1.3.9 provides a second example of how the external/internal use case categorisation can break down. Book excursion is a use case fragment from the perspective of the Customer, who can instantiate the use case as part of Make booking. To the Clerk, Book excursion is a specialisation of Modify booking and therefore a first order use case in its own right.

Projects have different stakeholders, who have different needs from the use case models they consume. From this perspective there is no right or wrong level of use case goal representation, but rather a requirement to see them being part of a hierarchical goal structure. This tension in goal expression has been addressed by Cockburn who has proposed a workable hierarchy that ranges through *Summary*, *User goal*, *Sub-goal* and *Too low*. Unfortunately, Cockburn does not provide extensive examples of his hierarchy at work, but does provide the foundation for navigating such a hierarchy based on goals where one form of expression is neither right nor wrong depending on the value offered to the model’s audience [50, 70]. This

hierarchy of goal decomposition will determine the number of use cases in a use case model, and accounts for differences of opinion. Evans warns against letting developers write high level use cases because they normally can not help themselves but to delve into implementation detail that is inappropriate during the early stages [61]. The need to be aware of both abstraction of detail and abstraction of goal expression is necessary to successfully produce use case models for all stakeholders when and where appropriate.

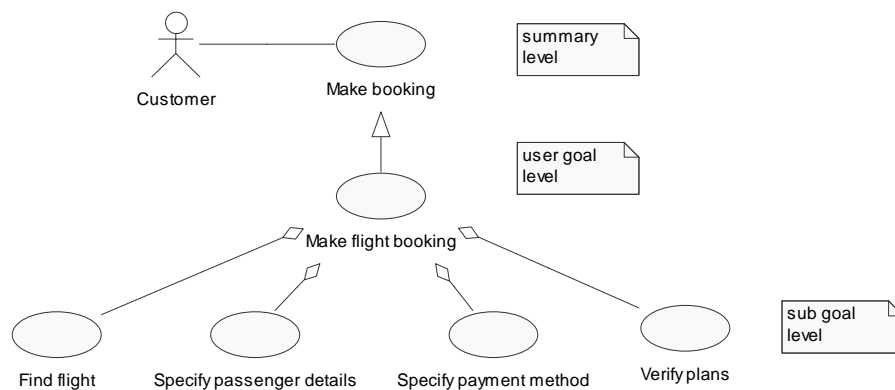


Figure 1.3.10: An example of Cockburn's hierarchy of abstraction of user goals in use cases. Strictly speaking, this type of association between use cases is not defined in the UML specification [63]. *Make flight booking* is shown as a specialisation of *Make booking*. *Make flight booking* is shown to comprised of a series of use cases that combine to represent a whole-part relationship.

To summarise, there is no agreement in the literature on the nature of abstraction in use case modelling, although there are sound approaches that can be employed to better understand the issues a requirements modeller must satisfy. Use cases are represented in a goal hierarchy in Figure 1.3.10. One of the major problems in use case modelling is deciding on the correct goal 'level' at which to represent requirements. The evidence suggests they are all correct, depending on the need of the target stakeholder. The other debate centres around the inclusion of either too much or too little detail, and the necessity to represent the use case set similarly in order to preserve some 'homogeneity of expression'. Use case modelling can be thought of as having different, inter-related, axis of abstraction. On one axis, the abstraction of increasing detail as articulated by Kulak and Guiney, and on the other axis, goal expression, as articulated by Cockburn. Together they provide guidance to build comparable and consistent models [71].

Use Case Modelling Summary

Since the day Brookes wrote identifying the non-existence of a silver bullet to put all the things right that are wrong with software engineering [72], practitioners have reacted with suspicion to any innovation that promised a holy grail. This holds true for use case modelling.

Glinz complains that by limiting inter-actor associations to exclude inheritance relationships the richness of the resulting models is excessively restricted, being unable to represent behaviour outside the system boundary. At times the specification simply seems incomplete as opposed to being deliberately restrictive. For instance, Glinz is critical of the actor model, with its emphasis on the definition of human roles representing the only mechanism that can trigger a use case execution. Indeed this is too restrictive where the modeller wishes to define automated behaviour.

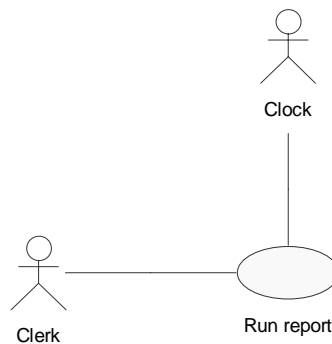


Figure 1.3.11: A *Clock* actor is introduced to represent the ability of the system to generate automated reports.

Figure 1.3.11 suggests that either the *Clerk* or the *Clock* can generate reports on an *ad hoc* or a scheduled basis. The concept of *Clock* actors that trigger behaviour is not defined in the UML specification [63], although it has been suggested as a useful addition by Cockburn [50]. (The *Clock* actor was first introduced in Figure 1.3.3.)

To some, the fact that use cases are not fundamentally an object-oriented notation is a particular issue [44], although because use cases are a notation for the representation of problems rather than solutions, this criticism may be unfounded. Given that use cases could be associated via inheritance and aggregation, use case modelling would become more compliant with an over-arching OO philosophy. However, drawing

any parallels between use case modelling and the eventual objects that are defined to satisfy the behaviour they represent must be resisted. Certainly use case modelling is founded on the principles of object modelling [59], which is sufficient, provided there is a recognised process for moving from the problem representation to an OO solution representation. To accomplish a mapping from problem to solution the interested reader is referred to the concepts embodied in the Model-View-Controller pattern [51] and the work that has been done subsequently [73] (*see*: Domain Objects on page 23). The specification does not expressly forbid the incorporation of useful modelling techniques such as inheritance and whole-part relationships between use cases. Certainly all of these concepts have been employed in this work (Figures 1.3.5, 1.3.7, and 1.3.10) which is testimony to some usefulness. Aggregation and inheritance are perhaps best suited to modelling the abstraction between use case layers in the goal hierarchy, whereas <<include>> and <<extend>> are sufficient for modelling at the layer where the use case has a text component associated with it. At this point a use case takes on detail and becomes less abstract.

Use cases do not describe NFRs. This has been levelled as a criticism, yet seems mis-placed, given that no claim was ever made for use cases to perform this role. Certainly NFR expression is important (the distinction between what is done and how it is done), yet NFRs are likely to hold true for many use cases and as such are more easily expressed somewhere else, such as in the definition of the underlying software architecture. Perhaps the solution to NFR representation can be found more meaningfully in the domain of design patterns. On the subject of use case text, Constantine and Lockwood reserve questions of use case execution to use case text (e.g. whether execution is strictly ordered). Glinz prefers to employ the notation introduced by Jackson [16] to represent sequence, alternatives, iteration, and concurrency. However, if *facade* use cases are, in effect, graphical artefacts of a table of contents, details of their implementation have no place in their graphic representation.

Most seriously, Glinz warns of the limitations of being unable to combine the concepts of state and state transition with use case models. Although state machines are available in the UML specification, they are not defined to be applied across more than one use case. Given that behaviour must depend on state it is not clear how a complete specification can be accomplished without reference to more global variables. As far as is known, no-one has tried to integrate state-dependent behaviour into a use case model until [62], albeit in a somewhat cumbersome manner.

The absence of more formality in use case definition and the paucity of comprehensive examples in use case modelling are unfortunate but inevitable given the newness of the approach. These shortcomings will inevitably be better addressed in the fullness of time. Other criticisms, such as the vagueness of the process for transformation from one model to another may have validity, but it is not so much aimed at use case modelling as at the totality of UML. The spectrum of criticism regarding UML more generally is, for lack of space, deemed out of the remit of this work.

It may be so that the use case approach is limited to the representation of database (transactional) systems [17, 18]. However, Jacobson's early work on telephone switches, and the increasing availability of real-time systems representation [74] suggests that use cases may be applied wherever it is possible to identify actors. Certainly the method is fundamentally based on the notion of actors which may act as an inhibitor where they cannot easily be identified. Use cases are simple to understand in principle, but that superficial simplicity masks significant issues of complexity.

Use case modelling has been available for sufficiently long to allow industry to have real project experience. However, the evidence as to its efficacy is scant. What evidence exists, suggests the effectiveness of use cases depends on how well the process is understood and implemented [75]. This paucity of reported evidence of success has not prevented use cases from becoming widely accepted in a remarkably short period of time [59], enjoying a take up that has been described as 'meteoric' [44]. They have achieved this by offering fundamental advantages over the alternative strategies that existed beforehand, even if, thus far, they have not delivered everything that might be hoped for them. Use case modelling is an effective means to improve the chances that a development team produces the correct system from the perspective of the functionality desired by the customer. On one level, use case modelling is an excellent method for the elicitation of 'black box' functionality, whereby the reader is interested in what the system delivers rather than how the functionality is constructed.

Although natural language is still an important feature of use case modelling, the problems of ambiguous representation can be reduced by adopting the strategy of writing them in an 'essential' (minimal, technology independent) form (*see*: Narrative style on page 26). Further clarity ('dis-ambiguation') can be attained by refraining from combining functional and non-functional requirements together in the same text

cases. Of their many advantages, use cases provide the 'glue' that stitches implementation objects together, filling the void in emphasising dynamic behaviour rather than static architectures [30].

Another point of view takes the position that a modelling language is only as good as the people who apply it. In the past, the phenomena of 'analysis paralysis' came to mean that there was the appearance of a lot of work taking place, but in reality nothing of particular value was being produced. The same risk exists with UML. A particular concern is to discourage management from coming to view UML as a panacea for all that is wrong in software engineering [76].

As the originator and a leading exponent of use cases, Ivor Jacobson believes that in future, work will proceed to establish links with the pattern community where the current set of design and implementation patterns will be supplemented by 'requirements patterns'. He advocates the employment of templates to establish a set of reusable use cases. In addition, Jacobson commented on the work of Gustav Karner on the subject of effort estimation in software procurement [32, 59], believing there is more to be done that could be of significant value in this poorly understood domain of software engineering.

Looking further ahead, Jacobson foretells a time when the development of bespoke applications will give way to the need to combine enterprise applications, packaged solutions, new applications and web services with legacy systems imagining new ways in which use case modelling will be applied. He sees more emphasis being put on *scenarios* and recommends that they should be treated with more formality and given greater prominence. In addition, the relationship between use cases and test cases should be streamlined; given a good use case should represent a good test case. At the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conference of 2002, Aspect Oriented Programming (AOP) was the 'buzz' topic. Jacobson is excited about the prospect of the complimentary nature of this new approach and use case modelling, where parallels between <<extend>> associations and 'aspects' were evident. In future, the integration of AOP and use case modelling could dramatically improve the way software is developed, paving the way for a new acronym; Use Case Oriented Programming (UCOP).

Requirements are less ambiguous when they are described with use cases. This suggests that if use cases were included in tender documents they could improve procurement by making requirements more clear and verifiable. The problem with including use cases in tender documents is that they take time to find,

define, and represent. Ways have been shown by which use cases can be produced more quickly depending on the level of detail that is required, highlighting the relative speed with which *facade* use cases may be constructed.

- [1.] Van Buren, J. and Cook, D., *Experiences in the Adoption of Requirements Engineering Technologies*, December 1998, Crosstalk, www.stsc.hill.af.mil/crosstalk/frames.asp?ure=1998/12/cook.asp
- [2.] Boehm, B., *Software Engineering Economics*, Advances in Computing Science. (1981), Upper Saddle River, N.J., Prentice Hall
- [3.] Zave, P., *Classification of Research Efforts in Requirements Engineering*. ACM Computing Surveys, (1997). **29**(4): p. 315-321, Associated Computing Machinery (ACM)
<http://citeseer.nj.nec.com/zave97classification.html>
- [4.] Lubars, M., Potts, C., and Richter, C.: *A Review of the State of the Practice in Requirements Modeling*. in *Symposium on Requirements Engineering*. (1992), 2-14, San Diego, California: IEEE Computer Society
www.cs.ucl.ac.uk/research/renoir/TBRC_RR01.pdf
- [5.] van Lamsweerde, A., Darimont, R., and Massonet, P.: *Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt*. in *Second IEEE International Symposium on Requirements Engineering*. (1995), 194 - 203: IEEE Computer Society
- [6.] Porter, A., Lawrence, G.V., and Basili, V., *Comparing Detection Methods for Software Requirements Inspections*. IEEE Transactions on Software Engineering, (1995). **22**(6): p. 563-575, IEEE Computer Society
- [7.] Abbott, R.J., *Program design by informal English descriptions*. Communications of the ACM, (1983). **26**(11): p. 882-894, ACM Press
- [8.] Spivey, J.M., *The Z Notation: a Reference Manual*. (1989), Upper Saddle River, N.J., Prentice Hall
- [9.] Khazaee, B. and Roast, C., *The Influence of Formal Representation on Solution Specification*. Requirements Engineering, (2003). **8**(1): p. 69-77, Springer Verlag
- [10.] Fowler, M. and Scott, K., *UML Distilled - A Brief Guide to the Standard Object Modeling Language*. (1999), Upper Saddle River, N.J., Addison Wesley Longman
- [11.] Jones, C., *Assessment and Control of Software Risk*. (1993), New York, Pearson Education
- [12.] Berry, D.M., *Software and House Requirements Engineering: Lessons Learned in Combating Requirements Creep*. Requirements Engineering, (1998). **3**: p. 242-244, Springer Verlag
- [13.] Herbsleb, J., D., Z., Goldenson, D., Hayes, W., and Paulk, M., *Software Quality and the Capability Maturity Model*. Communications of the ACM, (1997). **40**(6): p. 30-40, Association for Computer Machinery (ACM)
- [14.] van Lamsweerde, A.: *Goal-Oriented Requirements Engineering: A Guided Tour*. in *Requirements Engineering 2001 (RE01)*. (2001), 249-263, Toronto: IEEE Computer Society
- [15.] DeMarco, T., *Structured Analysis and System Specification*. (1979), Upper Saddle River, N.J., Prentice Hall
- [16.] Jackson, M., *System Development*. (1983), Englewood Cliffs, N.J., Prentice Hall
- [17.] Jackson, M., *Problem frames*, ACM Press. (2001), Harlow, Pearson Education Ltd., Addison Wesley
- [18.] Sutcliffe, A., *The Domain Theory - Patterns of Knowledge and Software Reuse*. (2002), London, Lawrence Erlbaum Associates
- [19.] Maiden, N. and Sutcliffe, A.: *Requirements Engineering by Example*. in *IEEE International Symposium on Requirements Engineering*. (1992), p. 104 - 111, Annapolis, MD: IEEE Computer Society

- [20.] Tracz, W., *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. (1995), Reading, MA, Addison Wesley
- [21.] Miller, G., *The Magical Number Seven, Plus or Minus Two*. Psychological Review, (1956). **63**: p. 81-97, American Psychological Association <http://www.well.com/user/smalin/miller.html>
- [22.] Gentner, D. and Stevens, A., *Mental models*. (1983), Hillsdale, N.J., Lawrence Erlbaum Associates
- [23.] Rosch, E., Mervis, C., Gray, W., Johnson, D., and Boyes-Braem, P., *Basic objects in natural categories*. Cognitive Psychology, (1976). **7**: p. 573-605, Elsevier Science
- [24.] Shank, R., *Dynamic memory: a Theory of Reminding and Learning in Computers and People*. (1982), Cambridge, Cambridge University Press (CUP)
- [25.] Meyer, B., *On Formalism in Specifications*. IEEE Software, (1985). **2**(1): p. 6-26, IEEE Computer Society
- [26.] Yourdon, E. and Constantine, L., *Structured Design*. (1978), New York, Yourdon Press
- [27.] Breuker, J. and Van der Velde, W., *CommonKADS Library for Expertise Modelling*. (1994), Amsterdam, IOS Press
- [28.] Beck, K.: *Patterns Generate Architectures*. in *Proceedings of the 8th European Conference on Object-Oriented Programming*. (1994), p. 139-149: Lecture Notes in Computer Science, Springer Verlag
- [29.] Hutt, A., *Object Analysis and Design - Comparison of Methods*. (1994), Object Modeling Group (OMG), Wiley Publishing
- [30.] Kulak, D. and Guiney, E., *Use Cases - Requirements in Context*. (2000), Upper Saddle River, N.J., Addison Wesley Longman
- [31.] staff, *Modular and Incremental Approaches to IT Delivery (Annex E)*, 2000, Office of the e-Envoy, UK Parliament, [http://www.e-envoy.gov.uk/oe/oe.nsf/sections/reports-itprojects/\\$file/successful_it.pdf](http://www.e-envoy.gov.uk/oe/oe.nsf/sections/reports-itprojects/$file/successful_it.pdf)
- [32.] Jacobson, I., Jonsson, P., Christerson, M., and Overgaard, G., *Object-Oriented Software Engineering - A Use Case Driven Approach*, ACM Press. (1992), Upper Saddle River, N.J., Addison Wesley Longman
- [33.] Merrick, P. and Barrow, P., *Testing the Predictive Ability of a Requirements Pattern Language*. Requirements Engineering, (2004): p. (Online edition at time of writing), Springer Verlag <http://springerlink.metapress.com/app/home/journal.asp?wasp=h07c7574wj2wnmf95ye0&referrer=parent&backto=linkingpublicationresults,1:102830,1>
- [34.] Pooley, R. and Stevens, P., *Using UML - Software Engineering with Objects and Components*, Object Technology Series, G.J. Booch, I.; Rumbaugh, J.: (1999), Harlow, Addison Wesley Longman
- [35.] Berry, D., Atkins, A.S., and Allen, P., *Managing Successful Projects with Prince 2*. (2001), Norwich, The Stationary Office (TSO)
- [36.] Jacobson, I.E., M.; Jacobson, A.; *Object Advantage - Process Re-engineering with Object Technology*. (1995), Upper Saddle River, N.J., Addison Wesley
- [37.] Alexander, I., Farncombe, A., and Mohammed, S.: *Towards better railway system specifications through scenarios*. in *The Railway Technology Conference (RailTex)*. (2002), Birmingham http://easyweb.easynet.co.uk/~iany/consultancy/railway_scenarios/railway_scenarios.htm
- [38.] Lauesen, S.: *COTS Tenders and Integration Requirements*. in *10th Anniversary International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ'04)*. (2004), Riga, Latvia: IEEE Computer Society <http://ieeexplore.ieee.org>
- [39.] Jones, C., *Conflict and Litigation Between Software Clients and Developers*, 2001, www.softerra.com/files/conflict.pdf
- [40.] Jones, K., *Who Signed that Contract? Inking Supplier agreements*, 12/11/2003, silicon.com, <http://management.silicon.com/itdirector/0,39024673,39116861-2,00.htm>
- [41.] Karner, G., *Resource Estimation for Objectory Projects*, 1993, Masters thesis, Linköping Institute of Technology, Linköping, Sweden, LITH-IDA-Ex-9344:21,
- [42.] Ribu, K., *Estimating Object-Oriented Software Projects with Use Cases*, 2001, Masters thesis, University of Oslo, Oslo, Norway, www.stud.ifi.uio.no/~kribu/oppgave.pdf

- [43.] Anda, B., Angelvik, E., and Ribu, K.: *Improving Estimation Practices by Applying Use Case Models*. in *4th International Conference on Product Focused Software Process Improvement, Rovaniemi, Finland, December 9 - 11*, pp. 383-397, LNCS 2559, Springer-Verlag. (2002), Profes, Finland
http://www.simula.no/publication_one.php?publication_id=500
- [44.] Firesmith, D.G., *Use Cases: the Pros and Cons*, (2002), Knowledge System Corporation (web),
<http://www.ksc.com/article7.htm>
- [45.] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language - Reference Manual*. (1999), Reading, MA, Addison Wesley
- [46.] Kruchten, P., *The Rational Unified Process - An Introduction*. (2000), Upper Saddle River, N.J., Addison Wesley Longman
- [47.] Constantine, L. and Lockwood, L., *Structure and Style in Use Cases for User Interface Design*, in *Object Modeling and User Interface Design*, p. 245-279 M. Van Harmelen, Editor. (2001), Addison Wesley, Upper Saddle River, N.J.
- [48.] Berard, E., *Be Careful with "Use Cases"*, (1995), The Object Agency, Inc., Germantown, Tennessee,
http://www.toa.com/pub/use_cases.htm
- [49.] Lilly, S., *How to Avoid Use-Case Pitfalls*, January 2000, Software Development,
<http://www.sdmagazine.com/articles/2000/0001/>
- [50.] Cockburn, A., *Writing Effective Use Cases*. (2001), Upper Saddle River, N.J., Addison Wesley Longman
- [51.] Krasner, G.E. and T., P.S., *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. Journal of Object Oriented Programming, (1988). 1(3) Application Development Trends (ADT), Framingham, MA
- [52.] Anda, B., Sjoberg, D., and Jorgensen, M.: *Quality and Understandability in Use Case Models*. (2001), 402-428, ECOOP 2001: Springer Verlag
<http://link.springer.de/link/service/series/0558/bibs/2072/20720402.htm>
- [53.] Schneider, G. and Winters, J., *Applying Use Cases - A Practical Guide*. (1998), Upper Saddle River, N.J., Addison Wesley Longman
- [54.] McMenamin, S. and Palmer, J., *Essential Systems Analysis*. (1984), Englewood Cliffs, NJ, Prentice Hall
- [55.] Kaindl, H.: *An Integration of Scenarios with Their Purposes in Task Modeling*. in *Proc. Symposium on Designing Interactive Systems*. (1995), Ann Arbor, MI, USA: ACM Press
- [56.] Graham, I., *Task Scripts, Use Cases and Scenarios in Object-Oriented Analysis*. Object-Oriented Systems, (1996). 3(3): p. 123-142, Computer Science Network (CompSciNet)
<http://www.dcs.kcl.ac.uk/staff/russel/CompSciNet/>
- [57.] Rosenberg, D. and Scott, K., *Use Case Driven Object Modeling with UML*. (1999), Addison Wesley Longman
- [58.] Biddle, R., Noble, J., and Tempero, E.: *Supporting Reusable Use Cases*. in *International Conference on Software Reuse (ICSR'02)*. (2002), p. 210-226, Austin, Texas: Springer Verlag
- [59.] Jacobson, I., *Use Cases: Yesterday, Today and Tomorrow*, IBM Developerworks, accessed: 2003, www-106.ibm.com/developerworks/rational/library/775.html
- [60.] Cox, K.: *Cognitive Dimensions of Use Cases - feedback from a student questionnaire*. in *12th Workshop of the Psychology of Programming Interest Group*. (2000), Corenza, Italy
<http://www.ppig.org/papers/12th-cox.pdf>
- [61.] Evans, G., Nazzano, F., *Why Are Use Cases So Painful*, 1999, Software Development (SD'99),
<http://www.evnetics.com/articles/Modeling/UCPainful.htm>
- [62.] Glinz, M.: *Problems and Deficiencies of UML as a Requirements Specification Language*. in *Tenth International Workshop on Software Specification and Design (IWSSD'00)*. (2000): IEEE Computer Society
- [63.] staff, *OMG Unified Modeling Language Specification Version 1.3*, 1999,
[ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf](http://ftp.omg.org/pub/docs/ad/99-06-08.pdf)

- [64.] Biddle, R., Noble, J., and Tempero, E.: *Patterns for Essential Use Cases*. in *Australasian Pattern Languages of Programming (KoalaPLoP)*. (2001), Melbourne, Australia
- [65.] Dictionary.com, Lexico Publishing Group, *accessed*: 2004, Los Angeles, CA, www.dictionary.com
- [66.] WordNet, Princeton University, NJ, *accessed*: 2004, Cognitive Science Laboratory, www.cogsci.princeton.edu
- [67.] Free On-Line Dictionary of Computing (FOLDOC), Imperial College Department of Computing, *accessed*: 2004, London, <http://foldoc.doc.ic.ac.uk>
- [68.] Smith, J., *The Estimation of Effort Based on Use Cases*, November 1999, The Rational Edge, <http://www.rational.com/products/whitepapers/finalTP171.jsp?SMSESSION=NO>
- [69.] Cantor, M., *Thoughts on Functional Decomposition*, April 2003, The Rational Edge, www.therationaledge.com/content/apr_03/f_functionalDecomp_mc.jsp
- [70.] Cockburn, A., *Structuring Use Cases with Goals*, *accessed*: 2003, <http://members.aol.com/acockburn/papers/usecases.htm>
- [71.] Merrick, P. and Barrow, P.: *Towards a Requirements Formalism in Procurement*. in *8th Annual Conference of United Kingdom Academy of Information Systems*. (2003), Warwick, England
- [72.] Brooks, F.P., *No Silver Bullet*. Information Processing, Elsevier Science, (1986)
- [73.] Spielman, S., *The Struts Framework - Practical Guide for Java Programmers*, The Practical Guide Series. (2003), San Francisco, Morgan Kaufmann Publishers
- [74.] Douglass, B., *Real Time UML: Developing Efficient Objects for Embedded Systems*. (2004), Upper Saddle River, NJ, Addison Wesley
- [75.] Lauesen, S. and J.P., V.: *Experiences From a Tender Process*. in *10th Anniversary International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ'04)*. (2004), Riga, Latvia: IEEE Computer Society ieeexplore.ieee.org/iel5/9273/29455/01335674.pdf
- [76.] Bell, A., *Death by UML*. ACM Queue, (2004). 2(1) American Computing Machinery (ACM) <http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=130>