WHITE PAPER

Visualizing Requirements in UML

Writing down your requirements in a formal, textual way ensures that they are well specified, and this often forms the basis for a contract between customer and supplier. However, there are times when being able to visualize those requirements might help both customer and supplier to gain a much quicker understanding of what is intended.
UML is a prevalent, graphical notation, much used by systems analysts and engineers. As such it is a widely known and easily understood notation.
This paper introduces part of the UML notation – Use Cases – and shows how they might be used to represent formal requirements in an attractive fashion.

Author: Ian Alexander

Version: 1.0

September 1, 2001

Capturing and agreeing requirements

Ultimately, systems are built to satisfy people's needs – for more efficient work, for faster travel, for easier home life, and so on. The people building the systems are in general not the people experiencing the need: there is therefore a communications gap between feeling a problem and developing a solution.

With small systems, this gap can be crossed quite readily, as the developers can discuss what is wanted with the users, and prototypes and early working versions can be examined and upgraded quickly.

With larger systems, such an informal approach often does not work, and all too many projects fail to deliver acceptable products on time and to budget. Numerous developers may be scattered over several sites; they may work for different companies, and are probably divided from the users by several layers of contract and subcontract, with little or no opportunity for direct communication.

The requirements clearly need to be documented so as to communicate the various viewpoints of the different kinds of user, the nature of the problem, and the shape of any acceptable solution. Conventional requirements documents typically contain either a large quantity of dataflow diagrams, which users often find forbidding and incomprehensible; or a large number of formally-written statements like 'The system shall enable the operator to select a contiguous portion of a document'. Readers often find these boring and may fail to see their relevance. Worse, in practice, any document made of many similar-sounding sentences is incomprehensible, as mistakes and omissions are hard to detect amidst the verbiage ('select an x', 'edit an x', 'delete an x', 'select a y', etc).

What we want, therefore, is a way of describing what users want, and then what systems should do to satisfy those needs. Both users and developers must be able to understand the descriptions quickly and easily. The requirements need to be in the users' language, but in a structure precise enough to guide development accurately. Then we have a chance to agree the requirements properly, because everyone knows what they mean.

Introducing UML and Use Cases

The Unified Modeling Language, UML, is a large and complex standard, allowing different kinds of people to build a wide range of system and software models ranging from high-level descriptions of business processes to precise definitions of the structure and behavior of software.

Fortunately, you don't have to know all about everything in UML to start using it for requirements. The construct that is most often used to hold requirements in UML is the Use Case. System specifications based on Use Case-modeled requirements often include other UML constructs including Activity and Sequence diagrams. We'll say more on these later, and get started with Use Cases.

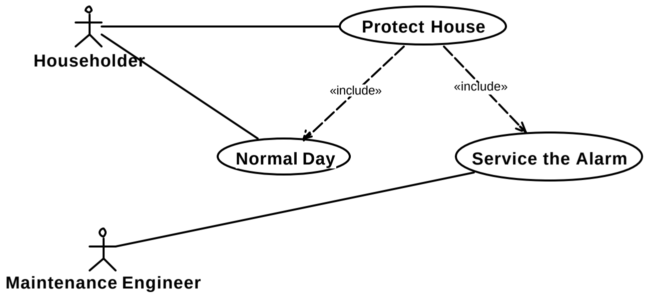
A Use Case is essentially a properly documented chunk of a process. It should have a single goal, and a basic scenario (a sequence of steps or activities) that someone can carry out to achieve the goal. Since life is never simple, there are always other ways of doing things, and things can go wrong, so a use case is not complete without a list of alternative paths and exceptions. If these are complicated, it is wise to make them into use cases in their own right, and to include them in – or otherwise attach them as extensions to – the parent use case.

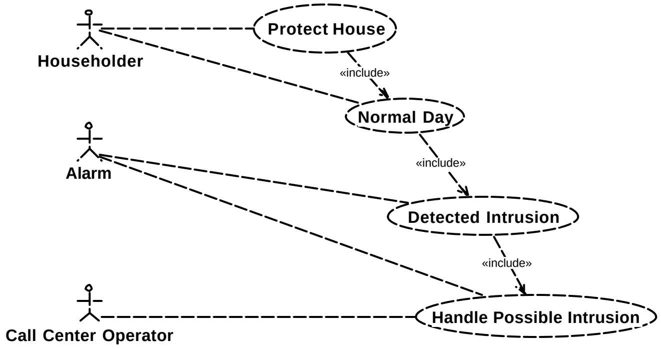
The use case notation was invented by Ivar Jacobson to describe how different actors interacted with and used some part of a large (existing) system, which explains why process chunks are known as Use Cases. However, the terminology is retained even when the cases are actually about the workings of a business, rather than of a hardware or software system. In other words, the approach works equally well whether you want to describe a problem in the user domain, or the behavior of a system that helps to solve such a problem. But it is definitely best to focus on one or the other at a time.

For the sake of example, suppose we are a household security company selling locks and bolts, wanting to get into the new and more profitable business of burglar alarms. Our business plan is to sell household alarms, and to make ourselves a steady income by servicing these regularly, as well as by monitoring the households centrally and calling out a guard when necessary.

We can begin by identifying the main actors involved. A Householder will buy one of our alarms to protect her house or apartment. The Householder uses the alarm each normal day to guard against possible burglary. A Maintenance Engineer visits once a year to service the alarm. And so on. In UML, each Actor is represented by default as a stick-man icon, whether the actor is a human or an active system (though we can customize this if we aren't satisfied with it). Each Use Case is represented as an elliptical bubble.

In the burglar alarm example, servicing the alarm contributes to the overall goal of protecting the householder's house, as an unserviced alarm may well fail. So, we can add an arrow to show that the 'Protect House' use case includes 'Service the Alarm'. It also includes the 'Normal Day' use of the alarm by the householder.





Getting Started: The Highest-Level Use Cases

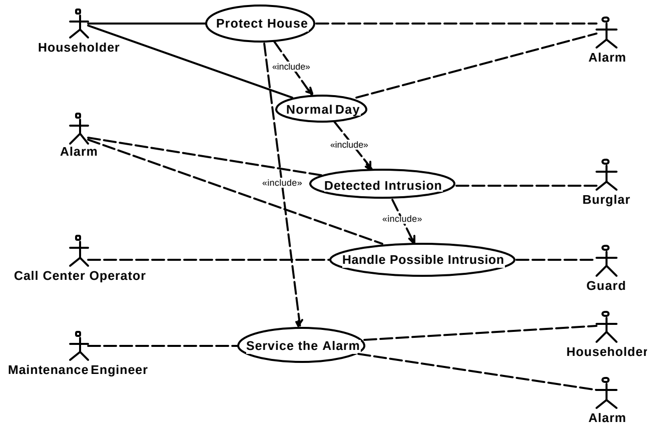
A straight line indicates that an actor is involved in that use case.
An arrow indicates that a use case includes another use case.

As we think through the business and the use of the system – for instance, in a workshop facilitated by an experienced requirements engineer – we can add more cases at both high and more detailed levels. It is convenient to show the high-level cases on the left, and successively lower-level cases towards the right. For example, the primary job of the alarm is to detect intrusion; it is up to the operator in the company's call center to handle it appropriately. Both cases are ultimately part of 'Protect House'.

Adding Detail: Including the next level of Use Cases

A use case on the left is at the highest level;
several levels can be shown on one diagram.

With a tool, we can add this extra detail, and then hide or show it as necessary. We can also choose which types of information to show – e.g., do we want to see all the actors, or only the primary actor for each Use Case? Here, for example, is a diagram summarizing all the actors involved in the normal use of the burglar alarm; other cases, such as breakdown and power failure, can be dealt with on other diagrams.

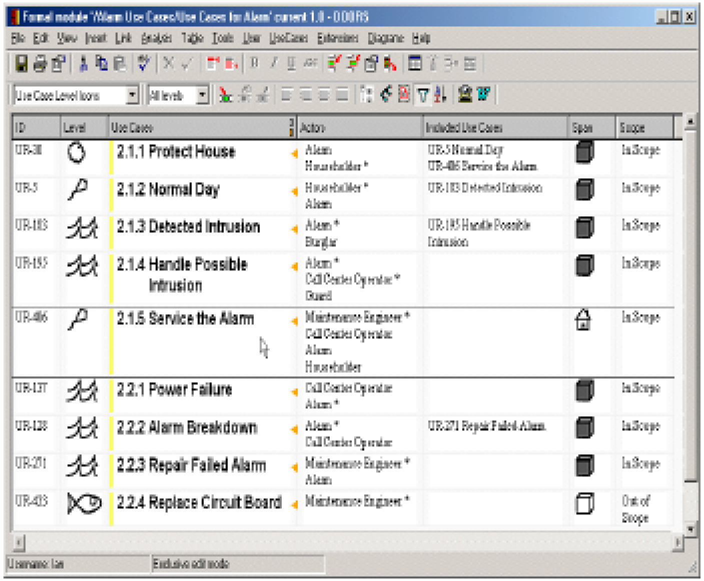


Use Case Diagram Showing Primary and Secondary Actors

Primary actors are listed on the left; secondary actors on the right. The actors are subclassed into human, system, and external agents, so here the stick-man icon has the special meaning 'human actor'. UML permits such customization.

Consider the Use Case 'Replace Circuit Board'. Should we be including it at this stage? It is certainly something that a Maintenance Engineer might do in the course of servicing an alarm. But if we are thinking about high-level cases like protecting a house and the business of servicing alarms as a whole, it is clearly a minor detail. We can follow the Use Case guru Alistair Cockburn and cut short the 'diving down a hole' behavior that often troubles requirement discussions, by drawing a simple and even humorous icon to indicate the level of the use case. 'Replace Circuit Board' is well below the surface for our purposes (though highly relevant when we come to consider the details), so we can draw a swimming fish icon. It is also looking inside the burglar alarm system (how do you know it is made of circuit boards?), so we can show it as a white-box use case. Once you have marked up a use case as being low-level like this, everyone will smile and turn their attention to the real business, which is to identify and describe all the major cases first.

Telelogic


Formal module '/Alarm Use Cases/Use Cases for Alarm' current 1.0 - DOORS

ID	Level	Use Cases	Actors	Included Use Cases	Span	Scope
UR-38	○	2.1.1 Protect House	Alarm Householder *	UR-5 Normal Day UR-406 Service the Alarm	■	In Scope
UR-5	🪁	2.1.2 Normal Day	Householder * Alarm	UR-183 Detected Intrusion	■	In Scope
UR-183	〰	2.1.3 Detected Intrusion	Alarm * Burglar	UR-185 Handle Possible Intrusion	■	In Scope
UR-185	〰	2.1.4 Handle Possible Intrusion	Alarm * Call Center Operator * Guard		■	In Scope
UR-406	🪁	2.1.5 Service the Alarm	Maintenance Engineer * Call Center Operator Alarm Householder		🏠	In Scope
UR-137	〰	2.2.1 Power Failure	Call Center Operator Alarm *		■	In Scope
UR-128	〰	2.2.2 Alarm Breakdown	Alarm * Call Center Operator	UR-271 Repair Failed Alarm	■	In Scope
UR-271	〰	2.2.3 Repair Failed Alarm	Maintenance Engineer * Alarm		■	In Scope
UR-433	🐟	2.2.4 Replace Circuit Board	Maintenance Engineer *		□	Out of Scope

Username: Ian | Exclusive edit mode

Cockburn-Style Icons for Use Case Level and Span

This tabular view summarizes the use cases in the project. Each use case has a defined level, indicated by an icon; a title (with all other text hidden); a list of actors, with primary actor starred; a list of included use cases; a span, which may be organization-wide (house icon) or focused only on a system (box icon), and which may be black - or white-box; and a scope (in or out).

Advantages of capturing requirements with Use Cases

Use Cases divide up the problem into bite-sized chunks, which everybody can understand, especially those people who are experts in the domain (burglar alarms) but who may not be expert in requirements engineering. Because the use case stories are easy to follow, they are more likely than conventional requirements to be checked carefully before development begins. This cuts down the risk of building the wrong system, and the embarrassment and cost of having a subcontractor come back to you having discovered deficiencies in your requirements. Equally, the subcontractor benefits from the confidence that the requirements are well understood.

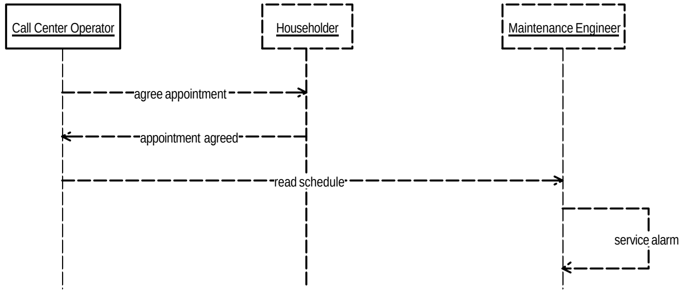
Use Cases are also a practical preparation for development, as UML is becoming the dominant approach for system analysis and design. In such a development project, modeling tools permit rapid navigation between use cases and associated specification and design diagrams. Such tools can check that definitions are consistent, and by switching between different models you can quickly build up an understanding of the system.

In short, Use Cases are a great navigational aid in a complex project, acting as index headings for the more design-oriented information. Handling requirements as Use Cases ensures a smooth and easy transition from problem to solution.

People often follow up on detailed system use cases (ones that go into what different parts of a sys tem do in turn to satisfy a need) by drawing UML activity diagrams with a 'swimlane' for each type of person or subsystem playing a role in the use case. This helps in the transition from specification to design, as the actors are natural candidates for design objects.

UML activity diagrams are similar to traditional flow diagrams, showing activities, decision points, and concurrency. The activities should correspond to steps or tasks in a use case, and the actors should already exist in the use case model, so a tool can check the consistency of the different models and facilitate refinement as necessary.

Another representation useful in system specification is the Sequence diagram. This shows (as its name implies) a sequence of messages passing between the participating objects. Receipt of a message implies that the receiving object has a responsibility to respond to the request. This allows depiction of different courses of action (or scenarios) that can be taken through the use case. Time flows from top to bottom of the sequence diagram.



Sequence Diagram used during System Specification
Time flows downwards. Each 'lifeline' shows activity sequence of one agent; interactions are shown as call-and-return arrows.

This approach leads directly to thinking about design – how does the call center supply a schedule to the maintenance engineer? Should the schedule be a design object? Does the engineer have a computer with a radio modem? These 'how?' questions move attention on from requirements to design. Conversely, 'why?' questions focus attention on essential requirements issues: why does the engineer have to read a schedule? How much of it must the engineer see? Making a satisfactory specification depends on being able to move freely in both directions, checking out questions of practicality as well as of the users' wants and needs.

Tools that allow easy navigation forwards and back between different models – say, Use Cases and Sequence Diagrams – can greatly speed and simplify the development process, as well as helping to give the users more precisely what they want. No-one ever came up with a quality product straight off: there is a vital iteration and set of trade-offs between what people want, what can be built, and what can be afforded. From an engineer's point of view, this is between the requirements, system specification, and design. From a manager's point of view, the trade-off is between quality, cost, and timescale. Either way, Use Cases are a firm foundation for reasoning about development.

Organizing requirements as Use Cases

All this is very nice and simple, but how does it help you organize your requirements? Alistair Cockburn argues that Use Cases are fundamentally a textual form:

"The trouble starts when you … believe that the diagrams define the system's functional requirements. Some people become infatuated with the diagrams… They try to capture as much as possible in the diagram, hoping, perhaps, that text will never have to be written...

The result, of course, was an immensely complicated drawing that took up more space than the equivalent text and was harder to read. To paraphrase the old saying, he could have put a thousand readable words in the space of his one unreadable drawing."

Writing Effective Use Cases, page 233

The Use Case diagram is, in short, a helpful summary list that shows the names of the use cases, their relationships and their actors: but it provides practically no detail on what each case consists of, beyond its goal (named in the title) and perhaps its main exceptions (if these are dealt with as separate use cases). In many systems, for instance, it is crucial that the designers know exactly what triggers each use case; this must be written down in the requirements.

There is no agreed standard for documenting use cases, but Cockburn, along with other authors such as Derek Coleman of Hewlett-Packard, have proposed templates for the purpose. The clear need is to describe the various paths of action as story-like scenarios, together with the circumstances in which they apply. These must be in a style that is clear both to engineers and to users of various kinds. Therefore the requirements must be written simply and without confusing detail.

Here then is a complete example Use Case for our burglar alarm, documented in full as suggested by Cockburn. It is a *business use case* as it describes the operation of the alarm company, rather than the mechanism of a system. It is *white-box* as it peeks inside the business. It is at *high level* because it avoids details like how the operator punches the keys to update a schedule. It is not particularly long; some examples in Cockburn's book run to several pages. Make your requirements as short as they can be, without being so terse that people misunderstand them.

2.1.5 Service the Alarm:

2.1.5.1 Primary Scenario

Call Center Operator arranges an appointment with the householder.
Call Center Operator schedules a Maintenance Engineer to service the alarm on the agreed date.
Maintenance Engineer reads the schedule on the agreed date, and travels to the householder's address.
Maintenance Engineer runs the standard diagnostic checks on the Alarm.

2.1.5.2 Alternative Paths

Householder contacts Call Center Operator to change the appointment. Call Center Operator updates the maintenance schedule.

2.1.5.3 Exceptions

Alarm is Faulty: Maintenance Engineer repairs the alarm and tests it again to ensure it is working correctly.
Alarm irreparable: Maintenance Engineer logs the problem, informs the householder that the alarm needs to be replaced, and makes an appointment with the householder for a return visit. The same Maintenance Engineer returns on the agreed date to replace the alarm.

2.1.5.4 Constraints

One-Star service contract holders are not guaranteed their choice of service date.

2.1.5.5 Trigger

A year has elapsed since installation or the last service.

2.1.5.6 Preconditions

Householder has a valid maintenance contract with the alarm company.

2.1.5.7 Stakeholders and Interests

Householder wants reliable alarm to provide security.
Alarm company wants regular service income.

2.1.5.8 Minimal Guarantees

Alarm operation is not disturbed by servicing.

2.1.5.9 Success Guarantees

Appointment happens on the agreed date.
Servicing is carried out regularly at the recommended interval.
Faults detected during servicing are handled promptly.
Serviced alarm works correctly.

A Fully-Documented Business Use Case

The heart of the use case is its title, which names its goal, and its primary scenario, which states how that goal is normally achieved, and which actor is responsible for each step. The other information fills in the details, setting conditions for success.

We certainly don't claim to be perfect at writing use cases; no doubt you can improve on this one. But it is simple and clear, and the headings force the writer to think – just a minute, what exceptions could there possibly be in this case? What triggers this case? Under what conditions can it start? Who has a stake in this, and what do they want? How will we know when the case has completed successfully? These are valuable questions, and they concentrate the mind wonderfully.

Controlling development

On a small and simple project, you may be able to document the use cases as just described using ordinary office tools. Word is fine if all you are doing is writing a specification with a hierarchy of headings and attaching it to a contract, and you can knock up the diagrams using a box-and-arrows graphics tool such as Visio.

But if your project is at all large, you will have to divide it into stages, and allocate the use cases to different developers or subcontractors; and you will need some at once, and some later. A solution comprising a requirements management tool with a UML modeling tool is very helpful for controlling such a project. It can among other things:

- Maintain traces between requirements and corresponding system specifications
- Maintain traces between requirements and acceptance tests
- Calculate metrics to indicate progress, and to highlight areas that need attention
- Draw up-to-date diagrams (such as use case summaries), guaranteed to be consistent with the requirements database
- Summaries the relationships of the use cases and actors
- Display tables of use cases to show status, scope, level, progress, and other attributes.

These are quite powerful advantages. In addition, if you – or your development contractors – are using tools such as Telelogic Tau to design the software, then you can benefit from automatic traceability and navigation between requirements, system specification, design, and test. Good requirements tools such as DOORS provide interfaces to many design, test, and project management tools, as well as application programming interfaces to allow new connections to be created readily.

Here, for example, are some metrics calculated for the current state of the burglar alarm project.

```
   Metrics for Use Cases in Module 'Use Cases
                   for Alarm'
-------------------------------------------------------------
Statistics:
   Use Cases:                          9
   Actors:                             6
   Primary Steps:                      28
   Alternative Paths:                  5
   Exceptions:                         8
   Local Constraints:                  2
-------------------------------------------------------------
Possible Problems:
   Use Cases with No Exceptions:       4
   Use Cases with Undefined Steps:     1
   Use Cases with Undefined Level:     0
   Use Cases with No Actors:           0
   Use Cases with No Primary Actor:    0
```

Simple Metrics on Use Cases

These metrics provide straight factual details on the use case model, and highlight possible problems with the model as it now stands.

Consider what these results mean. There seem to be about three steps per use case, which might be rather few but is certainly in the right ballpark. There are very few constraints and alternative paths, which suggests that not too much analysis of these aspects has yet been done. One case has no steps at all, and four have no exceptions at all, so these have obviously not been completed. However, all the cases have actors, so we know who is doing what. The project manager can get a clear impression of the state of the requirements, even though the metrics are quite basic.

Summary

Requirements in UML are clearer, easier to understand, simpler to assess, and offer better possibilities for control than old-fashioned specifications. Contrary to popular belief, they are neither arcane nor meant only for object-oriented software. They consist mainly of structured text, summarized by simple diagrams. Their sharp structure enables projects to be monitored using simple but informative use case metrics.

UML Use Cases allow you to visualize your requirements quickly and effectively. The resulting specifications remain mainly in plain English text, but are well-organized and can be understood immediately by both domain experts and engineers. UML Use Cases can be managed using ordinary office tools, but large and critical projects will certainly benefit from using professional requirements management and modeling tools such as Telelogic's DOORS and Tau products.

References

Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001, ISBN 0-201-70225-8
(see NewsByte at http://www.telelogic.com/newsbyte/ for a review of this book)

Telelogic DOORS (Requirements Management tool), http://www.telelogic.com/

Telelogic Tau (UML Modeling Environment), http://www.telelogic.com/

About the Author

Ian Alexander is an independent consultant specializing in Requirements Engineering and Business Process Modeling. He often works with Telelogic, providing consultancy and training on requirements using the DOORS platform.

His principal research interest is in improving the requirements engineering process by modeling business goals, processes, constraints, and scenarios. He is currently exploring the advantages of Use Cases on a technology project to investigate the reuse of specifications for control systems in the German automobile industry.

He helps to run the BCS Requirements Engineering Specialist Group and the IEE Professional Network for Systems Engineers. He is a Chartered Engineer.