ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/2807676

Structuring Use cases with goals

Article · December 1997

Source: CiteSeer

CITATIONS
264

READS
8,382

2 authors, including:

Alistair Cockburn
Humans and Technology, inc.

77 PUBLICATIONS **11,384** CITATIONS

SEE PROFILE

All content following this page was uploaded by Alistair Cockburn on 29 February 2016.

The user has requested enhancement of the downloaded file.

Structuring Use cases with goals

Structuring Use Cases with Goals[1]

Alistair Cockburn
Human and Technology, Norges Bank & Ifi/University of Oslo
email: arc@acm.org

Abstract

Use cases are wonderful but confusing. People, when asked to write them, do not know what to include or how to structure them. There is no published theory for them. This paper introduces a theory based on a small model of communication, distinguishing "goals" as a key element of use cases. The result is an easily used, scaleable, recursive model that provides benefits outside requirements gathering: goals and *goal failures* can be explicitly discussed and tracked; the goals structure is useful for project management, project tracking, staffing, and business process reengineering. The model and techniques described here have been applied and evaluated on a relatively large (50 work-year, 200 use-case) project.

Introduction

Jacobson's introduction of use cases [J92] immediately improved the situation that requirements documents often an arbitrary collection of paragraphs, having a poor fit with both business reengineering and implementation,. "Use case" is immediately attractive because the term implies "the ways in which a user uses a system". The term has caught on, and many people are using whatever they think of as use cases to good effect.

I have personally encountered over 18 different definitions of use case, given by different, each expert, teachers and consultants. They differed along 4 dimensions: purpose, contents, plurality, and structure.

Purpose: Is the purpose of use cases to gather user stories, or build requirements? (the values are <u>stories</u>, or <u>requirements</u>).

Contents: Are the contents of the use case required to be consistent, or can they be self-contradicting? If consistent, are they in plain prose or are they in a formal notation (the values are <u>contradicting</u>, <u>consistent prose</u>, <u>formal content</u>).

Plurality: Is a use case really just another name for a scenario, or does a use case contain more than one use case? (the values are <u>1</u> or <u>multiple</u>)

Structure: Does a collection of use cases have a formal structure, an informal structure, or do they form an unstructured collection (the values are <u>unstructured</u>, <u>semi-formal</u>, <u>formal</u> <u>structure</u>).

Clearly, if the purpose of using use cases is collect user stories, they may be self-contradicting, informal, and have no structure - and still be useful. So there are about 24 different ways to combine these into useful forms. I hasten to restate that they are useful, and that these forms already exist, are being used and taught. Therefore, I suggest that each person identify where their version of use cases sits in this space. The one this paper develops is:

Purpose = requirements
Contents = consistent prose
Plurality = multiple scenarios per use case
Structure = semi-formal

[1] This paper is being submitted for external publication. At time of publication, copyright transfers to publisher, from whom copies should be obtained. This version made available early for peer communication.

Why not just use Jacobson's form of use case? His are also **(requirements, consistent prose, multiple scenarios, semi-formal structure)**. As he put it (personal communication), he did develop a formal model for his use cases, but people could not accept or use the formal version. Therefore his published definition remains slightly vague, and the two-dozen variations sprang into life. The use cases I use in this paper are very close to Jacobson's.

The reason for doing any work at all use cases is that people on a project get really confused about how to write them. They request a little more theoretical guidance as to what to write, and above all, how to structure them. The reason that a semi-formal structure is used, is that it allows a formal underpinning that can be slightly abused, depending on what kinds of tools are available on the project. In other words, this is a theory intended to be used, and which is only justified by its usage.

In a further break with tradition, I present this paper, not as Introduction, Theory, Usage, etc., but as a sequence of problems posed when a project team starts using use cases, followed by a sequence of opportunities they discover as to how to use the use case structure to help on the project in other ways. These are the result of experience.

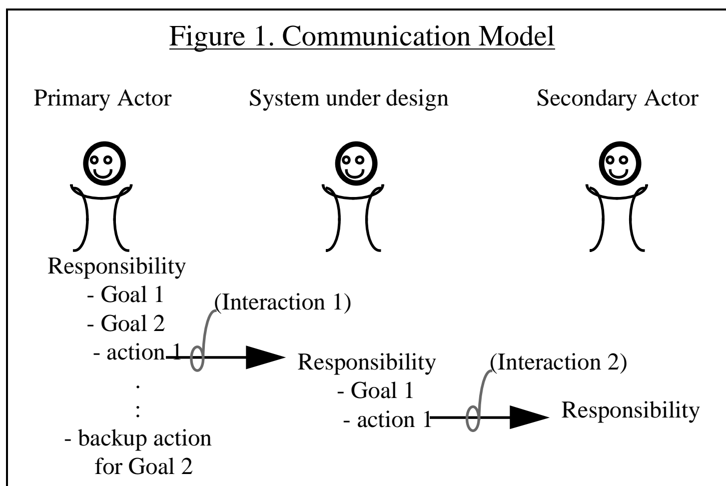
The theory was developed in early 1994, and applied on a 30-person, 18-month project having about 200 use cases. The team used the use cases, could navigate around them, could use non-specialized tools on them. The users could understand them. All the use cases were reviewed on-line as part of the contract requirements, and reviewed for project completion. The structure was used to estimate, staff, track and manage the project. The team said it was an improvement over how they had been working before and would use it again. In other words, the theory worked, but still has a few open problems, which are brought up at the end of the paper. It has been applied on other, smaller projects and taught in classes.

Problem (1). What are (these kind of) use cases?

A use case is a collection of possible sequences of interactions between the system under discussion and its external actors, related to a particular goal. This is the first, rough definition. Two more clauses are needed to complete the definition.

An action connects one actor's goal with another's responsibility.
The above definition identifies several things (see Figure 1). There are "actors", which may be people or computer systems. The system is treated as a single entity, which interacts with the actors. The use case will talk about sequences of interactions, and it will talk about variations in the sequences. To get farther than this, we need a model of communication and interaction.

Figure 1 shows a simple model of actor-to-actor communication through interaction. It turns out, trivially, that the system is itself an actor, and so the communication model need only work with actors. In Figure 1, the center actor is the system under discussion.



A *primary* actor is one having a goal requiring the assistance of the system. A *secondary* actor is one from which the system needs assistance to satisfy its goal. One of the actors is designated as the system under discussion.

Each actor has a set of *responsibilities*. To carry out that responsibility, it sets some goals. To reach a goal, it performs some actions. An *action* is the triggering of an interaction with another actor, calling upon one of the responsibilities of the other actor (see also Figure 3). If the other actor delivers, then the primary actor is closer to reaching its goal. The entity-relationship diagram corresponding to this is shown in Figure 2.

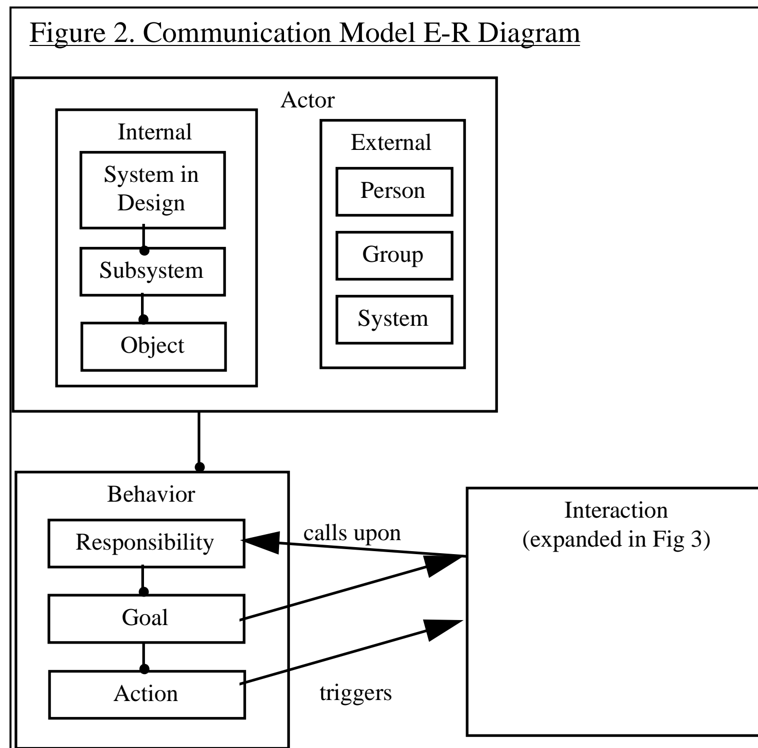


Figure 2. Communication Model E-R Diagram

Figure 2 says that there are internal and external actors. An external actor can be a person, a group of people or a system of any kind. The internal actor may be the system in design, a subsystem or an object. The system in design consists of subsystems, which consist of objects. Actors have behavior(s). The top-level behavior is a responsibility. It contains goals, which contain actions. An action triggers an interaction. The interaction is one actor's goal calling upon another actors (or its own) responsibility.

Goals are a normal part of our programming work, but no methodology or language yet tracks them formally. The developer writes the responsibility in the function name and comments, writes a comment that states the goal informally, e.g., "Find the path with the least cost", then writes the code that gives the actions. In terms of the above model, our development work goes straight from Responsibility to Actions, bypassing Goal. On the basis of the communication model, I hope that one day we shall have more formal tracking of goals and backup goals.

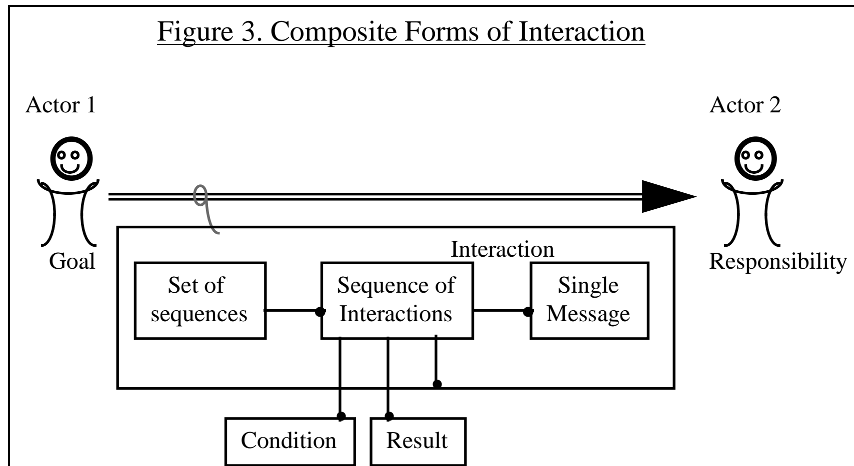
If the second actor does not deliver its responsibility, for whatever reason, the primary actor has to find another way to deliver its goal! This is marked as "backup action" in Figure 1. People are used to backup actions. Part of your job responsibility is to think up alternatives to reach a goal in case you suffer a setback. In programming, we use exception handling, **if** statements and return flags to reach the same effect. Goals and backups to failed goals are, however, not captured in programming languages - they are written in the programmer's comments.

The system, now having its responsibility called upon, start along a hierarchy of goals and actions. Goals have sub-goals and alternative goals, down to the level of individual

actions. One of the difficulties with use cases, as we shall see, is that fact that goals exist on many levels (which is confusing).

This communication model operates at all scales, from the largest organization down to individual objects. Broadcast and asynchronous communications are omitted from this model, but it suffices for very many situations. The use of semi-formal prose and semi-formal structure allows people to finesse the other situations in practice.

An interaction is simple or compound.
What is an interaction (see Figure 3)?



In the simplest case, an interaction is simply the sending of a message. “Hi, Jean” and “Print(value)” are examples of messages as simple interactions.

An interaction also could be a *sequence of interactions*. This is a recursive definition (as in Figure 3). At the bottom level, it consists of messages. We sometime want to bundle a sequence of messages into a single interaction item. This allows us to save space and effort. “I had an interaction with Jean in the cafeteria, today.” We refer to a sequence of interactions with one statement: “After you have your interactions with Jean this afternoon, call me with the results.”

A sequence has no branching or alternatives. It therefore is used to describe the past or a definite future, with conditions stated. Such a sequence is known as a *scenario*, which is how I also use the word. So “Sequence of Interactions” in Figure 3 is the same as “scenario”.

For the purpose of describing a system, we need to be able to collect together all the scenarios that might occur during one interaction. “Listen, when you have that interaction with Jean later today, here are some alternative situations you may run into...” or “An interaction with the banking machine may consist of the following scenarios.” A collection of scenarios is a *use case.*

A use case answers a question of the form, “How do I get money out of that banking machine?” You, the primary actor, have a goal, to get money out the system, which is that banking machine. There are a number of situations you could find yourself in. The use case collects them into one place. It breaks down your goal into subgoals, eventually individual message actions, plus the interactions between various actors as they try to reach that goal, or eventually fail to reach it.

Scenarios and use cases go until goal success or abandonment.
How do you know when to stop writing a scenario or use case? There are two clauses that have to be made explicit to get the proper bounds on a use case and a scenario. The clauses are the same for each:

Clause 1. All the interactions relate to the same goal.

Clause 2. Interactions start at the triggering event and end when the goal is delivered or abandoned, and the system's completes its responsibilities with respect to the interaction.

Clause 2 brings to the fore the notion of having a responsibility with respect to the interaction. Many systems have to log every interaction with a client (e.g., banking systems). If the clause did not include this requirement, the scenario would end without that responsibility being taken care of. Releasing a file handle is another example of this part of the clause.

Before giving the final definitions of scenario and use case, I note that the term "sequence" was used rather loosely above. "Single-rooted strict partial ordering" is the correct phrase. However, "sequence" is very close, shorter, and much more easily understood by the people about to write the use cases. When people ask, "What about messages that can happen in parallel?", the appropriate response is to say, that is fine, write it however you prefer or the tool permits. For brevity here, I say "sequence".

Scenario. A sequence of interactions happening under certain conditions, to achieve the primary actor's goal, and having a particular result with respect to that goal. The interactions start from the triggering action and continue until the goal is delivered or abandoned, and the system completes whatever responsibilities it has with respect to the interaction.

Use Case. A collection of possible scenarios between the system under discussion and external actors, characterized by the goal the primary actor has toward the system ís declared responsibilities, showing how the primary actor's goal might be delivered or might fail.

The characteristic information for a use case (see also Figure 3) is:

1. Primary Actor or actors
2. Goal
3. Scenarios used

The characteristic information for a scenario is:

1. Primary actor
2. Goal
3. Conditions under which scenario occurs
4. Scenario result (goal delivery or failure)

The scenarios are separated according to the conditions encountered, and grouped together as they have the same goal. The set of possible sequences is what constitutes system behavior.

Figure 4 gives an example of a scenario. It is short and simple, but obviously contains a great number of lower levels. The value provided by having an interaction described at this level is that it is easily reviewed and can be expanded when needed. Steps 2 and 4 lead down into the details of the insurance company's design. This particular interaction only happens in the case that I have a valid policy and accident details fall within policy guidelines.

Figure 4. Sample Scenario

System under discussion: the insurance company

Primary Actor: me, the claimant

Goal: I get paid for my car accident

Conditions: Everything is in order

<u>Outcome: Insurance company pays claim</u>

1. Claimant submits claim with substantiating data.

2. Insurance company verifies claimant owns a valid policy
   (failure here probably means goal failure)
3. Insurance company assigns agent to examine case.
4. Agent verifies all details are within policy guidelines.
   (an interaction between the agent and secondary actors)
5. Insurance company pays claimant
   (implies all preceding goals managed to pass)

The above construction of use cases and scenarios deals with the first two problems encountered when trying to use Use Cases: confusion over the difference between use cases and scenarios, and confusion as to what to include or exclude, when to start, and when to stop.

Problem 2: How do you control scenario explosion?
Scenario explosion has been a concern for many people who have worked with scenarios over the last several decades. Scenario explosion is avoided using three techniques: subordinate use cases, extensions, and variations. Subordinate use cases and extensions are discussed at length below.

Variations is a section of the use case text, which takes advantage of semi-formal structuring within and across use cases. Often, a use case at a high level uses input or output of one of several types. An example is payment by cash, check , credit, credit card, or electronic funds transfer. The differences between all those possibilities will have to be described at some point in a lower-level, but it would be wasteful to spend the time doing so at the higher levels.

We used a place holder, called Variations (see Figure 5), to track at the higher levels variations we know will occur. This is superior to creating either a myriad of high-level use cases for all the differences or a false input or output medium that pretends to be any of the actual ones.

It will have occurred to the reader that the steps in the scenarios are really intermediate goals of the actors. Each goal is attained by a sequence of steps or interactions, and might succeed or fail (or, in fact, partially succeed). This hierarchical composition of goals and steps forms a consistent, small algebra of composition, which is a way of saying, we get the pleasant gift that the way we like to work is supported mathematically. The structure helps avoid the explosion of scenarios that would occur if we were to try to simply list all possible ways of interacting with the system.

A sub-interaction is presumed to work.
A step in a scenario is written as though it works, e.g., “Verify that all accident details are within policy guidelines” (from the example in Figure 4). This allows the scenario to be written is a very readable style, with the most straightforward situation described simply and easily. The step is a goal, of course, which might fail.

Failure of a step is handled by another scenario, or an extension scenario. Many structuring techniques may be used to combine main and extension scenarios. From a broad, formal point of view they are all equivalent - they have the same expressive power, just different syntax and formats. Some people (the designers of ParcPlace’s Object Behavior Analysis) permit **if** statements and alternatives within a scenario. Some people like to rewrite the entire alternate scenario from the beginning, so each scenario can be read independently. Some people (myself and Jacobson) like to write scenario fragments as extensions to other scenario, to save writing and reading. A few people have observed that even flowcharts can be used.

It does not really matter which of these is used, except as which works best on the project, and which is best supported by the tools in use.

A use case collects together all the various scenarios. Figure 5 shows a full use case. The "Get paid for my car accident" use case, which contains the "Everything in order" scenario, will also have to contain the "Accident outside policy guidelines" scenario. That scenario will produce either an alternate success path, or a goal failure.

Figure 5. Sample Use Case

(System under discussion: the insurance company)

Primary Actor: the claimant

Goal: Get paid for car accident

1. Claimant submits claim with substantiating data.
2. Insurance company verifies claimant owns a valid policy
3. Insurance company assigns agent to examine case
4. Agent verifies all details are within policy guidelines
5. Insurance company pays claimant

Extensions:

1a. Submitted data is incomplete:

1a1. Insurance company requests missing information

1a2. Claimant supplies missing information

2a. Claimant does not own a valid policy:

2a1. Insurance company declines claim, notifies claimant, records all this, terminates proceedings.

3a. No agents are available at this time

3a1. (What does the insurance company do here?)

4a. Accident violates basic policy guidelines:

4a1. Insurance company declines claim, notifies claimant, records all this, terminates proceedings.

4b. Accident violates some minor policy guidelines:

4b1. Insurance company begins negotiation with claimant as to degree of payment to be made.

Variations:

1. Claimant is
   a. a person
   b. another insurance company
   c. the government
5. Payment is
   a. by check
   b. by interbank transfer
   c. by automatic prepayment of next installment
   d. by creation and payment of another policy

Use cases and scenarios make a recursive algebra

Each step in a scenario is a little use case! Just because a step in a scenario fails does not mean the use case fails - there may be a way to recover. An example is given in Figure 5. The insurance policy guidelines may been only slight violated (perhaps the client is some

sort of preferred customer), so a negotiation begins. Thus we get a small algebra for combining scenarios and use cases. See Figure 6 for the recursive containment of use cases and scenarios.

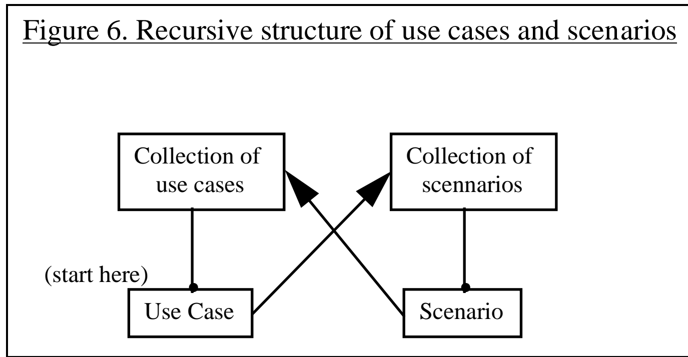


Figure 6. Recursive structure of use cases and scenarios

Figure 7 shows the structuring using the “striped trousers” image. The belt of the trousers is the goal that holds together all the scenarios. The stripes are the individual scenarios (each stripe characterized by its conditions and result). Each line in a scenario is a subordinate use case goal or a primitive action. A subordinate use case appears (often) in two stripes - once for should it succeed and once for should it fail. It may appear in more stripes if there are multiple recovery conditions. The stripes separate into two groups - those that deliver the goal (the “success” leg) and those that abandon the goal (the failure leg). No matter how many scenarios are in the use case, there are these two legs. In the rare cases of partial delivery, other legs may be added. Thus the entire use case may be inserted into higher level scenarios in the same fashion - as a success line and as a failure line (and as a partial success line, if that is a separate case).

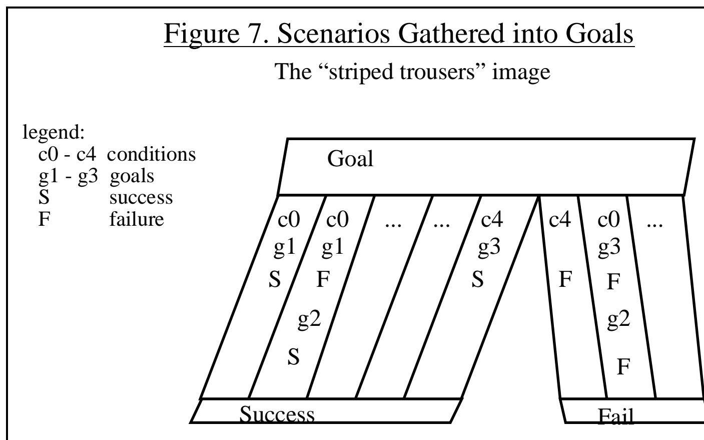


Figure 7. Scenarios Gathered into Goals

It is sometimes useful to distinguish the *main course* from the *alternate courses*. The main course is the simplest scenario, the one in which everything goes right and goal is delivered without difficulty - all the subordinate use cases succeed. It is the leftmost stripe in the trousers. It is possible, of course, that there are several possible simple, successful scenarios. To avoid formal entanglements here, it does not really matter which one or even if every one is called a main course. Using the structuring style in Figure 5, only one is written in full, and that is then the main course. Saying “the main course” means a scenario in which no recovery is needed.

The other paths that lead to success are the *recovery scenarios*. The paths that lead to goal abandonment the *failure scenarios*. The recovery and failure scenarios are collectively the *alternate scenarios*. The main course and the recovery scenarios all go on the success leg of the trousers.

Conditions get stricter down the scenario chain.
The explosion of scenarios is prevented because a subordinate use case contains and conceals a possibly large number of alternative paths. Thanks to the recursive technique just described, at each point in the scenario, everything that might possibly happen next gets reduced to a single pair - the next subgoal succeeds, or it fails. All the conditions that force one or another recovery scenario in the subordinate use case are concealed within the use case and need not be dealt with there.

What has to be distinguished are the conditions that separate success from failure for the subordinate use case. Thus, the outer scenario will name some conditions in the world, but ignore certain others. The inner use case's scenarios will introduce some new conditions that are relevant to those scenarios only. In other words, the list of conditions grows and gets more detailed as we proceed down the chain. This is as it should be.

In the case of a subordinate use case that is used by many higher-level use cases, it will probably ignore some of the conditions referenced by the higher-level use case.

It is perhaps worth restating that the actors usually do not know the conditions of the scenario at the start. The conditions are those that will drive the actors down a particular path. For example, a bank withdrawal scenario may have as conditions, "insufficient funds". I do not know this when I walk up to the ATM. However, that is the scenario that plays when I ask for more money than I have in the bank.

This completes the discussion of the structuring of use cases and scenarios. We now have: how use cases and scenarios relate, when to start and stop writing each, and how to structure the lot.

Problem 3: But users get lost in levels!

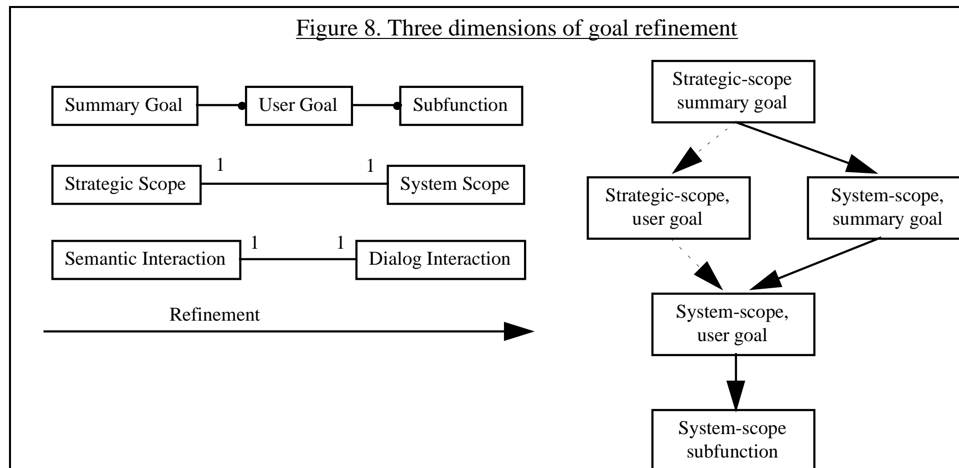
One of the sources of confusion for people is that use cases and scenarios work on many levels. There are even three sorts of levels, which triples the confusion. The first level involves *system scope* or boundary. The second level involves *goal specificity*. The third level involves *interaction detail*. To describe the levels in the three dimensions of refinement, I have to introduce some terms (see also Figure 8). The system under design has a primary actor having that goal. The boundary of the system under discussion here is the system being designed. The goal and the use case are at system scope.

That goal shows up in a larger system, usually the system that contains both people and the computer. Ideally, the goal shows up at the outermost edge of the project's control, the corporation (or organization). Finding the outermost level at which the goal appears is valuable, since it (a) shows exactly how the goal should benefit the organization, and (b) it is form most easily reviewed by the sponsoring executive. It is the level at which the system goal shows up as helping the project sponsors meet their goals. The goal and the use case are at strategic scope. The goal is a strategic goal with respect to the system.

One company using use cases discovered that their set of use cases addressed three different strategic scopes. One is for a customer buying goods: the strategic scope for the use cases is the corporation, and the primary actor is the customer (good idea!). A second is for the marketing department setting moneys aside for product promotions: the strategic scope is the IS department with computer systems, and the primary actor is the marketing department. The third is for the IS manager wanting to ensure security of the application: the strategic scope for the use cases is the entire computer system, the primary actor is the IS manager. Each set of use cases was written at the strategic level to ensure proper integration within the company, and refined to the system level to name detailed requirements. Separating these three boundaries was useful when reviewing the project as a whole, to discover who had vested interests in the system under design, and what the contribution of the new system to the corporation would be. It determined who should read the particular, high-level use cases. The outermost use case for customers buying goods took just over a page of text, and the sponsoring executive was able to find a small omission in it (which turned out to be outside this particular project) within a few moments.

The second dimension of refinement is *goal detail*. This is the dimension using the recursive structuring described in the previous sections. It consists of summary goals, user goals, and subfunctions (see also Figure 8).

Figure 8. Three dimensions of goal refinement



The level of greatest interest is the user goal, the goal of the primary actor trying to get work done. This corresponds to what might be called "user task", or "elementary business process". A user goal addresses the question: "Does your job performance depend on how many of these you do today?" "Log on" does not generally count as a user goal as it fails this test - logging on 43 times in a row does not (usually) satisfy the user's job responsibilities. On the other hand, "Register a new customer" is likely to be a meaningful user goal. Registering 43 new customers has some significance. We found that user goal often corresponds to a transaction in transaction systems (completing a transaction to the database).

Collections of user goals are summary goals. A summary goal may collect all of the user goals relating to product promotion, and be called, "Track promotions". These goals are useful for providing an index into a large set of user goals. Note that a user and summary goals may be written for the strategic scope or for the system scope.

Below user goals are subfunctions. A subfunction is a subgoal or step in an outer scenario, below the main level of interest of the users. Examples are logging on, and locating a product through a search. These are the subroutines of the use case family. Often a subfunction will correspond to a reused section of a screen: a dialog box or a page in a notebook.

A user goal may have a subgoal which is also a user goal. A common such situation is that of entering a new customer. One user-goal use case may be just entering the customer into the database, while another calls for doing that while processing a sale. The second contains the first as a subordinate use case, a line in a scenario. There is no problem with this, although it does tend to confuse some people.

The summary goals fan out to user goals in the form of a tree (bush, actually), and the user goals fan out to subfunctions in a criss-crossed way, forming a graph. The picture of all use cases by goal detail looks a bit like a sailing ship (Figure 9). We sometimes talk about the user goals being at sea level (the level that constitutes the project), with the summary goals looking like the sails and the subfunctions coming back together under the water.

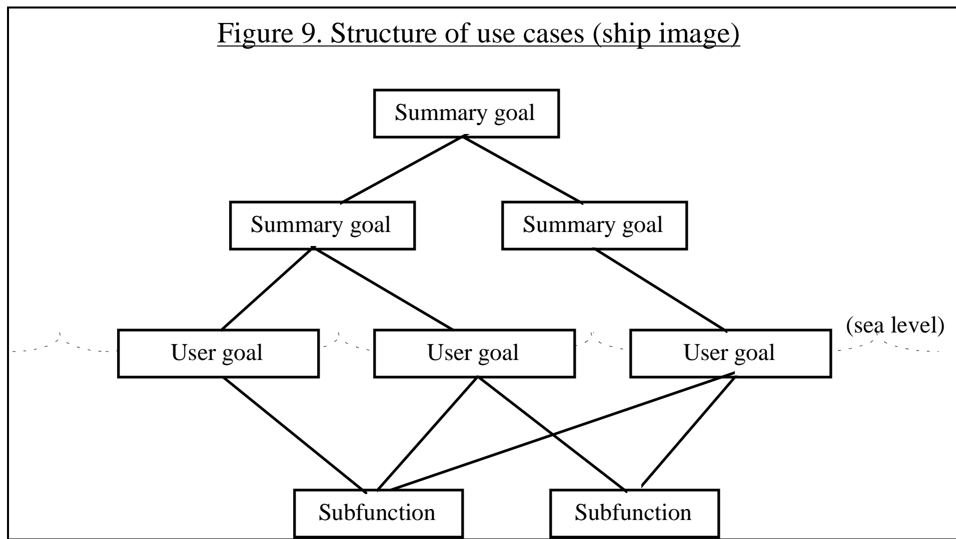


Figure 9. Structure of use cases (ship image)

The third type of level is *interaction detail*. In the example of entering a new customer's address, we may write, "enter address", or we may write, "enter street number, street name, suite number, city, state, postal code, country, ...". Even the goal, "move to the next field" would be valid. This lower level of interaction detail, when specific user actions, such as "tab forward" or" select from pull-down" list are named, are at the dialog interface level. I, and others, have found it most useful to stay away from the dialog interface level during requirements gathering. It is both time-consuming and subject to change when the final user interface is designed. The user interface design will address the dialog interface level.

Most people prefer to work at the outermost level practical, the semantic interface. In the address example, we would simply write, "enter address". The reason for choosing this level is to preserve the most latitude in (a) implementing the interaction in different technologies (such as voice or electronic interchange), and (b) accepting variations of the data (accepting different address formats for various countries, in the preceding example). The level of interaction detail is consistent across project, at least for user-goals.

Having these terms for the three dimensions of refinement facilitates discussion. It helps the people on the project extricate themselves from the confusion of use case goals being at so many possible levels. These terms also help clarify what various authors are doing with the use case concept. Larry Constantine "essential use cases" are system-level, user goals described at the semantic interface. This is a good and popular level at which to describe the system to be designed. When there are more than just a few use cases, more structuring is needed. That is when summary use cases and strategic goals are useful. Subfunctions are needed when there are some subfunctions that get used by a spread of use cases, and therefore must be tracked separately.

Only 7 of the 12 combinations of scope, goal and interaction combinations are useful, and only 3 or 4 tend to be used. Most people do not describe dialog interaction with use cases, although they could (they tend to use other drawings or prototypes). A useful sequence is to have the highest level use cases be strategic, summary goals, which decompose to system, summary goals, which decompose to system-level, user goals, which reference system-level subfunctions (see Figure 8).

Even with these distinctions and simplifications, just the fact that goals operate at so many levels is confusing. I come back to this as one of the "open problems".

Problem 4: Are these UI requirements?

Occasionally, people look at the definitions and examples of use cases in the literature and decide that use cases are just user interface (UI) descriptions. That is a valid interpretation from the definition of use case as "a sequence of transactions between external actors and the system." It is writing use cases at the dialog level of interaction.

There are two reasons to work a different way. Writing use cases just as UI descriptions deprives them of some of their value to the project. The design of the UI is likely to change too often for such writings to be used as contractual requirements and as system requirements. From the process or methodology perspective, the design of the UI comes later, after these goals and interactions have been named. Typically, a separate UI design team will come in, read the scenarios, and then play with different ways of presenting and collecting the information.

For those reasons, most people prefer to work at the semantic interaction level, describing just what information must pass the system boundary, without describing the sequence or nature of the interaction (without describing the dialog). To programmers, this corresponds to stating the parameters and result, as to a procedure.

Use cases written this way serve well as system requirements and as contractual requirements for the functioning of the system.

Problem 5: How do I write the text?

The use cases described in this paper have "semi-formal content" and "semi-formal structure". The purpose is to allow people to work quickly, with a variety of tools, and to allow them an escape hatch when the content or structure becomes too complicated. Making the content formal is too time-consuming for most projects, never earning back the benefit of the formality. Often, there is something tricky, something extra, something easy to describe in prose and hard to describe in a formal notation. Allowing people to use their own prose lets them communicate and get on with their jobs. Some use cases end up containing fragments of finite state machines, business rules, or temporal constraints. This is both good and bad. At the moment, the benefits from ease of writing and reading cancel out the disadvantages of imprecision and redundancy.

The general form of the prose that works well is:

<time or sequence factor>...<actor>...<action>...<constraints>

Thus, typical sentences might look like:

At any time after the clerk gets the quote, he may cancel the sale.

At the end of the month, she sends a credit memo to all customers whose credit is larger than a certain value.

After experimenting with writing styles, writers and readers agreed that they preferred to have actions numbered and starting on new lines, as in Figure 5. This keeps the narrative clear, improves traceability from requirements to design or test, and allows specific line references needed in the Extensions section.

The semi-formal structure allows the Variation section to be included. Since the variations are a list of input or output possibilities, a simple list for each suffices.

Problem 6: How do these terms match existing terms?

The key terms to match to are use case, scenario, script, main course and alternative course. Others are uses, extends, and subordinate (for use case or scenario).

A sequence of interactions from trigger to goal completion corresponds quite closely with many people people's interpretation of "scenario. This is in the spirit of Jacobson's writing, although he does not mention goals or goal completion explicitly. For many people, a key factor of a "scenario" is the absence of alternatives, which this paper's use of scenario preserves.

"A set of possible sequences that succeed or fail to deliver the goal" corresponds very closely to Jacobson's use cases, which is why I keep that term in this paper. The extension this paper provides is explicitly capturing of the goal. I did mention at the beginning of the paper that there are now 18 competing definitions of "use case", so you should compare definitions when using the term.

The leftmost stripe down the success leg of the striped trousers is Jacobson's "main course", the other stripes are his "alternate courses", the same terms I use. My extension is the separation of courses according to success or failure. A "course" is synonymous with a "scenario". A possible difference in the use of the words is that "course" refers to "what happens", where scenario includes both "under what conditions it happens" plus "what happens".

"Script" is a term from Object Behavior Analysis [RG92]. A script corresponds closely to what is called here a scenario. An OBA script differs mostly in some details of its formalism and its historical roots. OBA scripts allow alternatives, but that is mostly a syntactic variation, introduced to control the scenario explosion in a different way. Using the characteristics of the first section of the paper, OBA scripts are **(requirements, formal content, multiple scenarios, formal structure)**.

I called lower-level use cases "subordinate", the inverse of which is "superordinate" or "higher-level". Jacobson distinguishes the "uses" from the "extends" relation between use cases. I do not distinguish them in this paper, but in writing, the difference shows up (see Figure 5). Every line in a scenario names either a primitive action or a "subordinate" use case. This is the "uses" relation.

To save writing, rather than repeat the common part of a previously described scenario, it is useful to collect all the alternate courses under the heading "extensions". Each extension starts by stating where it picks up in the main narrative and what conditions are different. It then contains some lines and reverts back to the main narrative, or runs to completion on its own (for instance it might fail or complete a different way). This is exactly Jacobson's "extends" relation, but without the fuss. Whereas consultants, experts and users constantly exhibit confusion about the "extends" relation, users find the writing style in Figure 5 easy and natural. I simply never tell them the difference between extends and uses, and they do fine.

For those who do care about the difference, a "used" use case is a single line item in a scenario, which refers to another use case (a subroutine call). An "extension" is <u>another</u> scenario that refers to some point inside the scenario.

* * * * *

This completes the description of the problems and confusions surrounding use cases. The next sections describe some of the additional uses to which we have been able to put goals and the entire goal-phrase structure.

Opportunity 1: Attach non-functional requirements to goals

Goals provide a short catch-phrase with which to identify a functional requirement. But there are also non-functional requirements and other bits of information one wishes to capture, such as: What is required response time for this user goal? What is its frequency of occurrence of this goal, its peak rates and times? What is the relative priority of this function? Which users may use it? Which business rules does it rely upon? What is the relative size or complexity of this user goal?

The goals and the hierarchical structure in which they reside provide a useful way to organize these data. The information can be put at whichever level of the hierarchy best suits it. Obviously, user goals are the right ones to use much of the time, but complexity measures might also be attached to the subfunctions. In addition to attaching information directly to the use cases, we have used tables with the goals as the leftmost column, and almost any other kind of data in the other columns. We also have correlated user screens and class categories with goals.

Opportunity 2: Track the project by goals

The same goals that index non-functional information also provide the project management team a way to organize development and delivery of the system in increments. Again, the goals form the leftmost column in a table or on a project

management form. Teams are assigned to clusters of related user goals (which we have called "categories", as in, "user goal categories" or "use case categories"). We track progress of each user goal category with a stability measure that allows for iterative development of a user goal or goal category.

One of the things we have learned, working this way, is that the question, "When will the domain classes be complete?" is misleading. The domain classes are only ever complete with respect to the delivered functionality. The question we use instead is, "When will the domain classes be done for user goal so-and-so?" All notions of completeness are placed in reference to the goal creating the requirement.

Opportunity 3: Get subtle requirements from goal failures

The introduction of "goals" into the requirements activity has helped us discover early some of the more subtle requirements and scenario conditions. The requirements gathering or development person looks at each goal and asks, "How can this fail?" The person typically has enough imagination to come up with a number of interesting ways the goal can fail, which rapidly lead to discussion of some unthought-of situation or condition. In particular, this technique has helped people not terribly familiar with a domain to dig more deeply than before.

Goal failure can occur at any level. One of the more obvious questions is, "Suppose some data is missing from the database?" Although obvious, it is awkward, and has been known to get left out. Listing possible goal failures can help ensure that all these do, in fact, get addressed.

Opportunity 4: Use goals with responsibility-based design

Goals and scenarios have a good match to responsibility-based design [W90]. A goal is a request for service; a responsibility is a promise of service. Thus, the relationship between primary actor and system under design is that of client and server. The system's responsibility will typically include, but not be limited to, the delivery of the service. We have occasionally used refinement of scoping level to begin subsystem design and to document cross-platform design. Responsibility-based techniques help partition the system into subsystems. Afterwards, the original use cases are split up to build requirements for each of the subsystem. This can be carried out down to the level of individual objects, and is exactly what is happening when one draws an object interaction diagram for a scenario. Object Behavior Analysis [RG92], uses a similar zooming-in to the system components more rigorously than we do.

Opportunity 5: Match user goals to operational concepts

In comparing notes with requirements specialists and our own users, one comment keeps being brought up: this form of stating requirements describes what the user is doing with their own job. It is easier for the end users to react to the requirements specification. It also makes life easier for those writing the operations concepts document and those doing business process modeling or business process reengineering. Each function of the system is stated as a step of an outside actor. These steps are exactly what the process modelers need to do their work. We can trust that this use of goal statements will increase with time.

Experience Report: Mostly goodness, still a few gaps

What is badly needed is a scaleable theory for use cases that is easy to use. The ideas presented in this paper have been in use since early 1994, on medium-sized and smaller

projects. The theory holds, scenario explosion is avoided, and the users are comfortable reading, writing and working with the use cases. So the theory meets its goals. It still comes up short in a few places, notably, continued confusion about the levels, data variations, and partial delivery.

The difficulty with levels may be inherent with this way of building requirements. As was saw, there are three different ways of refining goals, which is an awful lot of complication. Introducing the terms “strategic goal”, “summary goal”, “user goal”, “subfunction”, “essential interface” and “dialog interface” helps somewhat, but not totally. Perhaps it is because these terms are still new that there is still discomfort, but I suspect it is simply because the technique is multilevel, and there is not a lot one can do about that.

Drawing the picture of the use cases is useful, to help avoid some of the confusion with levels. In the picture, all the user goals are placed at the same level to connote that level that is of value to the users. As mentioned, some user goals are also subgoals of other user goal, so not all of the user goals will, in fact be at the same level. We have used shading to indicate user-level goals, with good effect.

The second open problem is in working with data format variations (see Figure 5). In the structuring technique described in this paper each subordinate goal case carries forward the list of variations, until it is time to break them out into their own use cases. This works, but is still a bit unsatisfactory and *ad hoc*.

The third problem is that a goal may not delivered in its entirety. There is an additional dimension of composition not mentioned above, that of <u>features</u>. Rating an insurance policy may depend on what other policies or insurable items the policyholder has; valuing an order may depend on the pricing agreement structure in place. These are features of the situation or features of the actor. During system development, the simplest feature is developed first, and the other ones rolled out over time. Thus, a goal or use case is delivered for a given set of features. Often, the features cross multiple use cases, making the tracking more complicated.

Each feature could be written in a separate goal case and scenario. Besides being tedious, it violates the idea of working from the user’s point of view, where there is a single goal (“rate the policy”, “value the order”). It is the state of the data that causes the difference. We have tried introducing the notion of feature to split use cases, but it has been unsatisfactory. The cost of not having a good solution is that project management is more complex.

Experience with the goals has been almost uniformly good, subject to the items just mentioned. One project had approximately 230 use cases. About 150 of these were user-level goals, perhaps 50 of the remainder were for shared subfunctions like “find customer” or “enter address”. About 30 goals were introduced as summary goals to help manage the user goals. There were 1-4 levels of summary goals between the top goal, and 0-3 levels of subfunctions. There was considerable confusion about levels until the leveling vocabulary and the iceberg picture were introduced. This form of requirements structuring has been examined by a number of people from different backgrounds, including requirements gathering and business process modeling, and they have generally been pleased with the form.

References

[RG92] Rubin, K, and Goldberg, A. “Object Behavior Analysis”, in Communications of the ACM, vol. 35 no. 9, (Sept. 1992).

[J92] Jacobson, I. et al. Object-Oriented Software Engineering: A Use-Case Driven Approach, Addison-Wesley, Reading, MA, 1992.

[W90] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, NJ, 1990.

View publication stats