Patterns for the Role of Use Cases

Gertrud Bjørnvig
Microsoft Business Solutions
(Formerly Navision)
Frydenlunds Allé 6
DK - 2950 Vedbæk
gertrudb@microsoft.com

This paper contains a set of patterns for the role of use cases in the software development process. Use cases are seen as a mean to capture functional requirements in a way that makes it possible to drive and coordinate the work of the different roles in the team – done by having the use cases as a common center for all team members. That means that the focus for the patterns are more related to the usage of use cases than to the more technical areas related to how you write a good use case. In that sense the patterns can be seen as a supplement – or maybe a superstructure – to Steve Adolph and Paul Bramble's "Patterns for Effective Use Cases".

Introduction

Use cases are a well-known technique, but it varies from project to project why and how the technique is used, and what advantages development teams achieve from using the technique. I hope these patterns can help teams to release the full potential of their investment in use cases.

The target audience of this paper is people who do have some experience with use cases. If you need an introduction to the more technical part of how to write good use cases, I can recommend "Writing Effective Use Cases" by Alistair Cockburn [6], and "Patterns for Effective Use Cases" by Steve Adolph and Paul Bramble [1].

Use cases are proposed as an excellent tool to capture functional requirements, but not the only one. Maybe you only need stories as recommended by Extreme Programming [3]. Maybe you need supplemental techniques – for example decision tables – to get a good overview and structure for your functional requirements. I do not advocate for classical requirement specifications with numbered requirements that are mapped to use cases, though I know that some organizations do have procedures that require numbered requirements in addition to the use cases.

This paper does not cover the question when use cases are the most appropriate technique, and when they are not. And it doesn't cover when you need supplemental techniques. It could be an idea for a future version of the patterns – to give a more whole picture of the role of use cases.

My starting point for working with use cases was the few pages about use cases in Jacobson's book from 1992 [9], and his general idea of a "Use case driven approach". Years later, it was nice to read Cockburn's book "Writing Effective Use Cases" [6], that confirmed many of the insights I had got during my work with use cases.

Though I still stick to Jacobson's basic idea about a use case driven approach, I haven't experienced that the modeling approach with traceability through the models (that is essential for Jacobson) is the most effective part of the use case technique. Focusing on modeling and diagramming often results in too detailed and complex models that don't give the necessary overview. I have found Cockburn's "Structuring the Use Cases with Goals" [5] as a more essential approach – yet hard to do. What I see as important for use case driven development is not so much the traceability from a modeling point-of-view, but more the strength of the use cases as *the* central items that all roles in a development team can use as a center for their work. It can give another kind of traceability – for example, from a use case to its user interface (UI), or from a use case to its test cases.

The patterns are captured during my work with use cases in different companies since the mid-nineties. Most of the examples and stories are from the Danish company, Navision – a part of Microsoft Business Solutions since the summer 2002. I am referring to *Navision* in the paper, despite that the name of the company has changed. Primarily because the things referred happened when the name was Navision. But it is actually still correct since *Navision* now is the name of the product line where most of the referred use case experiences are captured.

Five of the seven patterns have been workshopped at VikingPLoP 2002 (KNOW-HOW KICKOFF, NARRATIVE AND GENERIC, DEVIATIONS DEFINE SCOPE, GOALS DEFINE NUMBER, and USE CASE AS CENTER).

In addition to Steve Adolph and Paul Bramble's "Patterns for Effective Use Cases" is the patterns also related to some of Jim Coplien and Neil Harrison's "Organizational Patterns".

Terms

I'm using the following terms in the paper:

Core use cases: Use cases that are NOT derived from other use cases by using the UML-based kind of relationships: "Extend", "Include", or "Generalization".

Main scenario: A use case scenario with the steps providing the normal sequence of actions. Also called "Basic Flow of Events" [11], "Main Success Scenario" [6], "Sunshine Scenario", or "Normal Sequence".

Deviations: Everything that deviates from the main scenario, for example, extensions, failure handling, exceptions, and variations.

Form and style

The patterns are written in Alexandrian form [2], and the pattern names are written in SMALL CAPS.

Pattern Overview

This paper includes the following seven patterns:

KNOW-HOW KICKOFF: The use case work can begin too early or too late – so do a use case session that works as a KNOW-HOW KICKOFF before designing, but after the initial market research and envisioning.

READINESS REFLECTION LIST: It is hard to know when a use case is finished – so maintain an open issue list that can help you to decide if the use case is finished enough so you can begin another activity, for example design.

NARRATIVE AND GENERIC: Use cases can be too detailed or too abstract – so separate detailed story telling from functional requirements by having a narrative as an introduction to the use case.

GOALS DEFINE NUMBER: Too many use cases mean lack of overview – so limit the number of use cases through goal-oriented use case definition.

DEVIATIONS DEFINE SCOPE: It can be hard to estimate time and resources based on the use cases – so optimize the knowledge of the use case scope by listing all the deviations you can imagine.

USE CASE AS CENTER: Often use cases end up being a one-man or one-role show – so let the use cases be the teams' center for feature development by organizing other work products around the use cases.

SIZE THE ITERATIONS: Iterations can be too long or too short – so base the iteration plan on 1-3 use cases per iteration in no more than 5 iterations.

The above list also provides the natural sequence; however there are other relations of the patterns as well (see Fig. 1).

It is good to establish a READINESS REFLECTION LIST from the beginning of the use case work, meaning that the first version of the list will be a part of the result of a KNOW-HOW KICKOFF. The READINESS REFLECTION LIST is an important tool during the whole process, so placing it in the beginning of the sequence is only to indicate that you need it from the beginning.

To apply the pattern SIZE THE ITERATIONS, it is a good idea to have USE CASE AS CENTER. It will mean that you can plan iterations that not just include the work of the developers, but also include the work of for example testers and user documentation. To have USE CASE AS CENTER you need GOALS DEFINE NUMBER and DEVIATIONS DEFINE SCOPE to be able to manage the scope.

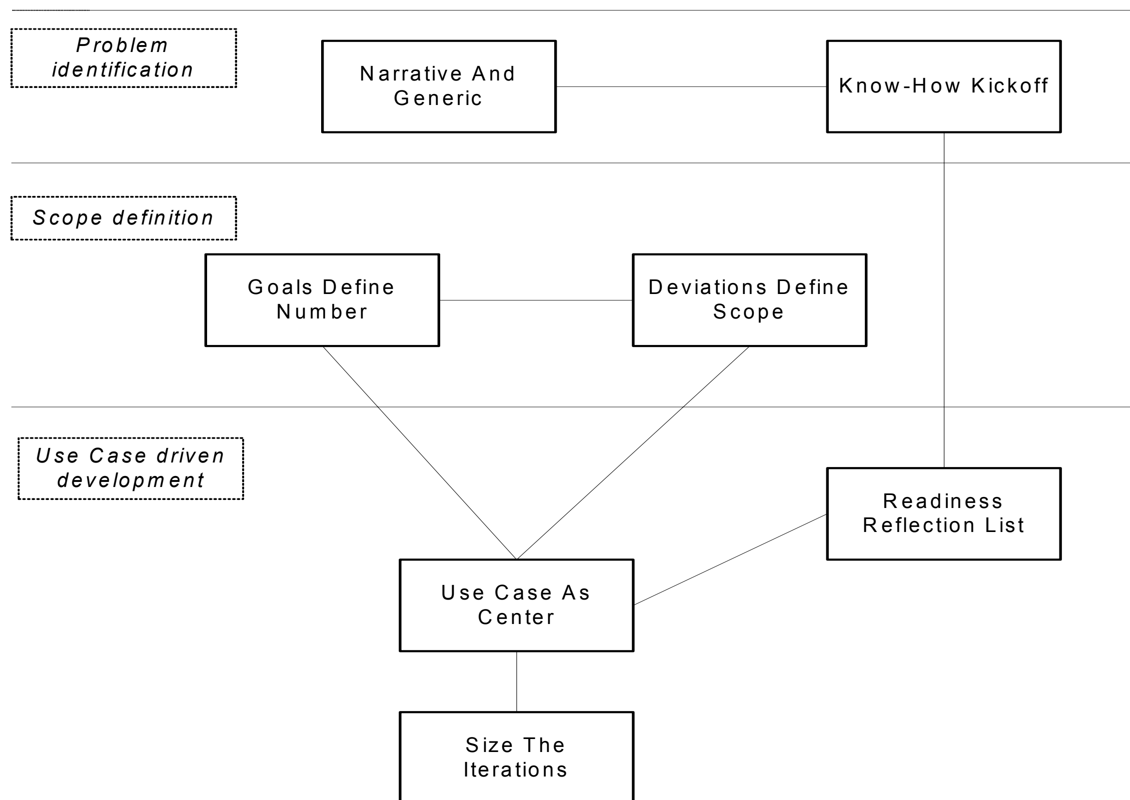

Fig. 1: Pattern Overview

The problem identification could be done by other techniques. But a very good starting point for the more detailed use case work is to have an overall use case diagram showing the actors and the most important use cases (the result of a KNOW-HOW KICKOFF), supplemented by a set of stories to show the intention of the use cases from a user's point-of-view (a part of NARRATIVE AND GENERIC).

GOALS DEFINE NUMBER and DEVIATIONS DEFINE SCOPE help you to manage your functional requirements in the context of a users work situation. These patterns are the heart of the use case technique. They support you in focusing on goals, finding an appropriate abstraction level (which still requires experience and hard work), and to manage the scope.

Depending on the type of development you are involved in, and the preferences of the development team, you can choose a more or less "Use case driven approach" [9]. The three patterns USE CASE AS CENTER, SIZE THE ITERATIONS, and READINESS REFLECTION LIST are all essential if you want to do use case driven development.

The Patterns

Know-How Kickoff

… the team is organized. The initial project overview is established, and you have decided to apply use cases in the project.

***

If we begin the use case work too early, we don't know enough to identify valuable use cases. If we begin too late it can be hard to get the team's attention and motivation for doing use cases.

Sometimes when we begin applying use cases we are guessing more than identifying and analyzing. We can only define one actor "the user" because we really don't know who the actors are. Or we define detailed and trivial use cases such as "Update Customer Information" or "Set up Vendor Type" instead of focusing on the real problems we want to address. Maybe we haven't been able to involve people with sufficient knowledge or maybe we're trying do use cases too early. The result is use cases that aren't substantial enough to encourage further exploration.

We can also begin too late, then it can be hard to get team members' attention and participation, as I realized in this situation:

> The whole team is gathered to a use case session. A drafted design exists. The project manager and the usability specialist think that lack of use cases is a problem for the project. The developers are eager to begin coding. The product managers are eager to see results. They feel that they know what they are going to develop and cannot see any reason for doing use cases now. Most of the session is spent on discussing why we should do use cases now.

It was too late for the team. It was obvious that most of the team members had a clear picture of what they were going to do – but they didn't have the same picture! And they didn't want anything or anybody to cloud this picture; "I know what to do, and I don't want anything that can delay progress now."

If a team does not have a consensus about the purpose of the project, it can be very hard doing use cases. The lack of consensus can be an obstacle for the work – especially later in a project.

So we have to get the team's attention in time, and we need to identify the use cases that really matter.

Therefore:

Before starting any design activities: gather persons with know-how about market, customers, users and domain to a joint use case work session. Identify the most important actors and use cases. Base the session on a product vision or business case.

A well-timed use case work session – with qualified know-how represented – is an effective way to ensure that we identify the important use cases from the beginning. And it is a good kickoff for the subsequent use case work. Well-timed means when you know why you are doing the project and how you expect your users to benefit from it, but before you have begun any design activities.

At Navision we had a product manager role as a part of the team. The product manager represents the customer, and it is crucial for the success of a KNOW-HOW KICKOFF to have this role present. This role is similar to a SURROGATE CUSTOMER [7] role where you ensure that you fill the role with someone who will try to think like a customer in cases where a real customer isn't available.

It is preferable that the majority of the participants are project team members – it can give the nice side effect of a mutual buy-in to the use cases. It could also be other stakeholders involved; this would support the principle of PARTICIPATING AUDIENCE [1] where you involve customers and internal stakeholders in the use case work. But avoid having too many people in the session – no more than ten. It can be very effective with just three people, if they are the right people. It is project-dependent how you get the best mix of people for a KNOW-HOW KICKOFF.

Document the session with an overall use case diagram showing the actors and the identified use cases.

During the session is it also a good idea to list all open issues and questions, instead of trying to resolve everything right away. It reduces the time used on discussions, and it prevents too much guessing. Such a list will be your first version of a READINESS REFLECTION LIST.

My experience is that with the right people – representing sufficient knowledge about the project – in the room, you can identify the 5 to 12 most important use cases for a project in a few hours. Of course, this depends on the number of people and the scope of the project.

***

UNITY OF PURPOSE where "the leader of the project must instill a common vision and purpose in all the members of the team [7]" is normally a precondition for a successful KNOW-HOW KICKOFF, but it can also go the other way – the preliminary use case work can help achieve UNITY OF PURPOSE.

KNOW-HOW KICKOFF uses the principle of HOLISTIC DIVERSITY [7]. It gathers a few people with diversified skills and lets them communicate directly. The same goes for DIVERSITY OF MEMBERSHIP [8] where the principle is used for requirement teams. The Navision feature team model supports these patterns very well; the team core

roles (product manager, developer, tester, user assistance, usability, project manager) are gathered in the same team from the beginning of a project. The team is sitting physically close to each other. The Navision feature team model is based on the team model described in Microsoft Solutions Framework [12].

There is also an element of LOCK 'EM UP TOGETHER [7]; we need to gather different people in the same room to do a very focused task.

A KNOW-HOW KICKOFF can also work as a kind of working FACE-TO-FACE BEFORE WORKING REMOTELY, where you "begin a distributed project with a face-to-face meeting for everyone [7]." A typical distributed project involves geographic distance and different time zones. But even a short geographic distance as the one between buildings and floors can have a challenging impact on the communication. Not to mention the mental distance among roles belonging to different organizational units with different value set – for example business roles and development roles. A project involving subcontracting is another example of a distributed project where a KNOW-HOW KICKOFF can work as a kind of working FACE-TO-FACE BEFORE WORKING REMOTELY.

KNOW-HOW KICKOFF is a good beginning, but it doesn't ensure further progress in the use case work in itself. The next challenge is to write the use cases. One way to handle this is to establish a SMALL WRITING TEAM [1] who is writing NARRATIVE AND GENERIC use cases with SCENARIO PLUS FRAGMENTS [1]. SCENARIO PLUS FRAGMENTS means that you write the main scenario without any considerations of possible deviations, and below it, you list the deviations. Review their work through a variation of GROUP VALIDATION [7] with focus on requirements instead of design. It can be done as a TWO TIER REVIEW [1] where the review process is divided into two types of review: first one or more smaller reviews with few people involved, and then a group validation with the complete group. Don't forget that DEVIATIONS DEFINE SCOPE – you don't know the scope before you have a clear picture of possible deviations, for example related to failure handling.

Readiness Reflection List

… you have begun the use case work.

***

The work with use cases can go on forever, but it is hard to define criteria for when the use cases are finished.

When our use cases have been through a couple of iterations of work, we begin to wonder when we can finish the work with use cases and go on with other activities. It seems like the use cases will never been good enough – there will always be room for improvement, the use cases will never be complete.

On the other hand, we don't want to dig too much into for example the design work, if we have big unresolved issues related to the use cases.

We want the use cases to be stable, not static. The use cases shall reflect the actual changes, but we would also like to move on with other activities as design, code, and test.

What we need is sufficient use cases, not finished use cases. Use cases that are sufficient to begin another activity. Maybe one version of the use cases is sufficient for beginning some prototype work, but not sufficient to begin writing test cases. We need to know when we are ready to do what.

Therefore:

Capture all open issues on a READINESS REFLECTION LIST. Maintain an overall list for issues that goes across use cases – plus a separate list per use case for use case specific issues. Evaluate the issues to know if you are ready to begin a new activity.

The nature and number of open issues gives us a good picture of the situation. Is it a big issue? Is it a minor issue? How long is the list? The open issues at the list can help us to reflect about our readiness. Look at these issues and you will probably know if you are ready; ready to do a prototype, ready for design, ready for writing test cases, ready for any kind of activity that you want to be based on use cases.

It requires a good habit of constantly parking questions and open issues on the relevant list. Go through the list and clean up regularly. The list should be a common responsibility for all team members, and not a specific project management tool.

***

Making it a habit to park open issues on a list instead of trying to solve everything at once, can make many meetings more effective. This example from Navision shows how open issues lists were used in a workshop context:

> Domain experts from several countries were gathered to contribute and agree about some complicated new functionality that we wanted to apply to our product. A draft set of use cases were the common starting point for the one-week workshop. The first day we did a walkthrough of the use cases, and every time a question or open issue occurred, it was written down in an open issue section in the use case description. I participated the first day and noticed that it was very big open issues that were mentioned. Some of them were questioning the whole purpose with the functionality. I returned at the last day of the workshop, and noticed that now there were much more issues in the open issue sections. But these issues were small, more detailed and specific – it seems like the big issues had been worked out during the week. I don't think that I could have seen this progress in the work just by reading the updated version of the use case scenarios. And it was clear after the first day that the big open issues had to be resolved before any other activities related to this functionality could begin.

It can be hard to get useful feedback from stakeholders, including customers, on a set of written use cases. It is much easier to get feedback on concrete open issues.

Using a READINESS REFLECTION LIST can help you QUITTING TIME where you "stop developing use cases once they are complete and satisfactorily meet audience needs" [1]. Completeness is difficult to define, so the principle of a READINESS REFLECTION LIST is that the open issues define the completeness: "No unacceptable open issues – then it must be complete".

Narrative And Generic

… use cases capture the functional requirements. The core use cases have been identified in a KNOW-HOW KICKOFF, but not yet described.

***

Often use cases are too detailed to be useful as functional requirements, or they are too abstract to be understandable.

Business people tend to have a lot of details in their use case descriptions. They like to tell the story.

Domain experts tend to write more abstract, sometimes too abstract, use cases. Everything is so familiar that things become implicit.

Developers tend to begin design in the use case description. They are focusing on the solution.

It is important to have a common picture of what we are going to develop, and a generic use case description do not give us vivid pictures as stories do.

We love stories and details, but we want precision and brevity.

Therefore:

Begin the use case with a narrative that illustrates the intention of the use case. The concrete details that are good in a narrative can then be separated from the more generic functional requirements where too many details are disruptive and cause imprecision and inconsistency.

The use cases can concisely describe functional requirements and at the same time, we can get a clear picture of a real context of use. This can be seen in the following example:

> Part of narrative:
> Christian finds the customer through the handheld device and checks out the name of the contact person at the company, John Jensen. The last time he visited John, he was just about to celebrate his 25th jubilee. Christian wants to ask about that at the meeting.
>
> Corresponding part of use case:
> The sales person checks information on contact person.

***

To make qualified narratives you need sufficient data about the users' situation. The usability domain offers several good techniques of how to gather valid user data through field studies.

My experience from use case workshops is that it can be hard for a team to describe a use case from scratch. But when we start with a narrative we get a common picture that makes it easier for the team to describe the more generic part of the use case. I have not seen it work the other way around; when the generic part of the use case has been written it seems to be very hard to tell the story.

One team in Navision describes their vision as a story presented as a role-play and documented with slides. It was easy to derive use cases from that story, since everyone had a common picture of the idea before the use cases were identified.

Many use case descriptions in Navision are supplemented with a storyboard as the narrative. Most narratives include at least one deviation from the main scenario. Narratives can also go across use cases.

Narratives can give life to actor descriptions as well. Describe a real user by giving a user profile with age, name, tasks etc. to supplement the more generic actor descriptions. *Personas* serve the same purpose by inventing an artificial person.

NARRATIVE AND GENERIC can help us write PRECISE AND READABLE use cases that are "readable enough so that the stakeholders bother to read and evaluate it, and precise enough so that the developers understand what they are building [1]."

Goals Define Number

… a number of use cases have been defined.

***

We often end up with a lot of detailed use cases which isn't useful for design and test.

It is hard to find a right and consistent detail level for use cases. This goes both for the detail level for the use cases as a whole, and for the detail level in each use case description.

It is also tempting to think that we can cover everything in the use case descriptions. We cannot – we need additional description techniques to document all requirements for a system.

If the use case technique is used as a decomposition tool, the number of use cases will easily increase to too many, meaning that we miss the overview. Sometimes use case diagrams look more like dataflow diagrams.

If the use case descriptions contain many details, for example related to user interface or system components, it is hard to get an overview, and almost impossible to keep the use cases updated.

Therefore:

Keep only those use cases that support the user in meeting their work-based goals and the corporate business goals. Discard use cases that is only a mean to achieve a goal, but isn't a goal in itself, such as "Log On" or "Update Customer Data". Be specific on WHAT the user wants to do, and WHAT the system is responsible for – and not on HOW it is done. This will help you finding the right detail level.

This is not easy and requires experience and good examples!

An actor-goal list a la Cockburn [6] – with both primary actors (users) and actors representing the business (stakeholders) – can help you validating that a use case support an actor's goal.

You can also prioritize your use cases in order to identify the FOCAL USE CASES [3]. It is the use cases that are most important to users, but the prioritization also includes other parameters, such as stakeholders' interests and risks expressed by development.

Focusing on actor goals help us to reduce the number of use cases, and it helps us to deliver useful software!

***

Finding the right user goal level for a use case can be hard. One project in Navision handled it this way:

> The overall goal is formulated as a positioning statement: “Returns Management transforms customer dissatisfaction into customer satisfaction”. One of the use cases that support this statement is “Register Compensation Agreement with Customer”. This use case represents an important goal for the user and the business. A part of this use case is to “Register compensation agreement for a special item going to be repaired by the vendor”. It is tempting to model this as a separate use case, but this would probably result in a decomposition mode, where we begin to describe every single deviation as a use case itself. If we do so we would maybe need 50 use cases to support the positioning statement instead of the 8 that we had in this project.

Having sub-goals like this at the step level in the use case can help us manage the number of use cases. I define steps broadly as steps in the main scenario plus steps representing deviations from the main scenario, including extensions. Very few extensions need to be described as full use cases. A row in the deviation list will often be sufficient.

To solve the problem of too many CRUD (Create/Retrieve/Update/Delete) use cases in administrative systems, you can use parameterized use cases as described by Alistair Cockburn [6]. Alistair Cockburn was also the first to describe the principle of goal oriented use cases [5].

GOALS DEFINE NUMBER helps you to constrain your number of use cases. A controllable number of use cases are a precondition for having USE CASE AS CENTER. My experience is that a project should have no more than 15 use cases for a 6-9 month time period with a team size at no more than ten people. See also SIZE THE ITERATIONS.

Deviations Define Scope

… the core use cases have been identified and are characterized by GOALS DEFINE NUMBER. The use cases are NARRATIVE AND GENERIC with a defined main scenario.

***

You don't feel that you can base your time and resource estimations on the use cases, or it has taken much longer to implement a use case than expected.

How do you estimate your use cases? This question can actually mean two things: How do I estimate? Or: How do I ensure that the use cases provide a good foundation for estimation?

If it is the first question, the use cases cannot help you. It is just as hard to estimate use cases as anything else.

The second question is more relevant. We feel that the use case is finished when we have written the main scenario. It represents the core functionality, so we think we can go on with design and implementation. But we realize that the main scenario is a minor part. All the extra things that the user should be able to do in this context must be implemented. And all the things that can go wrong must be taken care of, too. Sometimes these things have an impact on the design that we haven't even been aware of. So at the end of the day, we have spent much more time on implementing all the things that have *not* been a part of the main scenario than on the main scenario itself. If we aren't aware of it in beforehand we have a very poor basis for estimation.

A use case with nothing but a main scenario – no extensions, no exceptions, and no variations – is only a beginning. The main scenario is the framework for deviations and represents in many cases less than the half of the scope. Each step in the main scenario is a potential source for several deviations.

The strength of the use case description structure is related to the relation between the main scenario and the deviations. This structure allows us to handle complexity in a simple and consistent manner. But it requires that we are able to define the "normal" scenario – without any "if's" at all.
Many use case authors have a hard time doing that, but my experience is that it is always possible to define the normal scenario – and it is necessary in order to get structured use case descriptions. This is the principle of the pattern SCENARIO PLUS FRAGMENTS where the solution is to write the main scenario "as a simple scenario without any considerations for possible failures. Below it, place story fragments that show what alternatives may occur." [1]

At the same time we can't know everything before we start coding. Some things will not be discovered before we begin to test. But we still have to optimize our knowledge about the scope to estimate and prioritize. We have to be able to prioritize in order to meet our schedules.

Therefore:

Define the potential scope of the use case by listing the deviations from the main scenario in one or more deviation identification sessions. For each step in the main scenario ask questions like: What can go wrong here? What else does the user want to do here? What is happening if…? List the answers in the deviation section of the use case.

The deviation section can include variations, extensions, error handling, or exceptions – depending on the need for being able to distinguish between the different kinds of deviations. Keep it as simple as possible. The completeness of the list is more important than the categorization.

Involve people that fulfill different roles in order to get a qualified list of deviations. A tester is good at focusing on things that can go wrong. A usability expert is good at focusing on user experience. This is the principle of HOLISTIC DIVERSITY [7].

The deviation list provides a realistic feeling for the potential scope of a use case. It can help us decide if we want everything on the list in scope or not, and we improve our basis for estimating.

***

At Navision, we have had good experiences with gathering domain experts in a workshop where a first draft of the use cases – including the main scenario – is used as a basis for structured deviation identification.

EXHAUSTIVE ALTERNATIVES has a similar solution: "Capture all alternatives and failures that must be handled in the use case [1]." But the focus here is more to avoid that the developers will misunderstand the system's behavior, so the system will be deficient. DEVIATIONS DEFINE SCOPE focuses more on how to manage the scope. Anyway; achieving both can only be good!

Sometimes it isn't worthwhile doing the deviation identification for all use cases at once. It is sufficient to do it for the use cases for the next development iteration. This is closely related to SPIRAL DEVELOPMENT where the use cases are "developed in an iterative, breadth-first manner, with each iteration progressively increasing the precision and accuracy of the use case set [1]."

Use Case As Center

… the team has a shared understanding of the goals and the scope of the project achieved through KNOW-HOW KICKOFF, NARRATIVE AND GENERIC, GOALS DEFINE NUMBER and DEVIATIONS DEFINE SCOPE.

***

Use cases have the potential to be the repository of important agreements about functionality, but often they end up being a one-man or one-role show or they just die.

It is a challenge to keep existing documentation updated. Especially if the document has been a part of an approval procedure, where the goal more is to get the approval than to have a useful document during development. But outdated use cases are not useful for coding, testing, or user documentation purposes.

Use cases are often used in the beginning of a project to capture agreements about direction and then they die. This can be okay and has a value in itself. But it means that it can be hard to know what is being implemented and why. Often the developers are the bottleneck in a project, and the only one who is updated about important decisions related to functionality. It can make the work of other roles – for example testers – hard to do. And it can result in expectation mismatch with the customer, because important decisions are taken in the wrong context – for example during coding.

I have heard statements like "Use cases are only for developers" or "We only do use cases in order to do test cases". If the use cases are written for a specific role – and probably by that role, too – this role, and nobody else will use them. The use cases are written in a way so they are only useful for this role.

Other statements like "I cannot test from these use cases" or "It isn't possible to design from the use cases" are signs of a role that hasn't been involved in developing the use cases, but now is expected to use the use cases as a basis for their work. Involvement is everything.

Therefore:

Let the use cases be the center for other work products: organize test cases around use cases, insert references between for example use case and user interface, list tasks in relation to use cases, and plan the development iterations based on use cases. Involve both stakeholders who are expected to use the use cases (team members) and stakeholders who represent the customer in the use case work.

If other work products are organized around the use cases the team will have a common interest in having readable and updateable use cases.

When use cases synchronize the work of the team, they become the natural driver of the development iterations.

Use case driven development is an effective way to control iterative and incremental development, and if you succeed doing it you will get the full benefit of your investment in use cases.

***

Having USE CASE AS CENTER can help preventing the problem of deceased use cases, but it can be very hard to repair the situation.

I have only seen real use case driven projects when the leader (formal or informal) of the team wants it like that. It has to be planned that way.

Architecture and system design will go across use cases and is not naturally linked to use cases. Some projects need a more technology and architecture driven approach, but the use cases will still be a good way to connect the development work with test and user documentation (a user doesn't have to be an end user – some times the user is another developer).

In FEATURE ASSIGNMENT features are assigned to people for development, but it is emphasized that this should be coupled with the role of CODE OWNERSHIP (each code module in the system is owned by a single developer) to "strike a balance between maintaining architectural integrity and getting the job done [7]." The same is true for USE CASE AS CENTER; you need a balancing mechanism to ensure architectural integrity.

Use case driven iterations can be used to avoid designing more than what is actually needed for a given iteration.

ADORNMENTS [1] have similarities with USE CASE AS CENTER by recommending that nonfunctional information should be associated with the use cases in supplementary sections.

There is a relation between USE CASE AS CENTER and WORK FLOWS INWARD. WORK FLOWS INWARD means "Work should flow in to developer from stakeholder, especially customers. Work should not flow out from managers [7]." The use cases are an agreement between stakeholders and the team about the functionality of the system. When we have USE CASE AS CENTER, most of the work can be derived from the use cases instead of being generated by a manager role. It means that USE CASE AS CENTER can support that WORK FLOWS INWARD.

The principle in USE CASE AS CENTER is essentially the same as Ivar Jacobson's "A Use Case Driven Approach" [9] [10]. While Jacobson is advocating for a modeling approach with traceability through the different models, USE CASE AS CENTER is focusing more on use cases as a common center for the team's work.

Size The Iterations

… GOALS DEFINE NUMBER and DEVIATIONS DEFINE SCOPE. You have USE CASE AS CENTER and want to do iterative and incremental development based on use cases.

***

We invest a lot in our use cases, but when we begin to code we don't use them.

We base our implementation on design and tend to base our implementation plans on design more than on functionality.

Many organizations are still struggling with the waterfall approach that often results in "big bang" testing.

Iterative and incremental development is a challenge to manage. Often will development and test be less and less synchronized for each iteration.

Therefore:

Make an iteration plan with one to three use cases per development iteration, and no more than five iterations. Prioritize the content of the iterations with respect to customer needs and architectural risks. Ensure that the iteration transition criteria cover both code and test.

The plan can result in an external or internal release. What matters is that you have something that is finished enough to be shipped.

***

A project where GOALS DEFINE NUMBER of use cases, will typically have 8 to 15 use cases for a 6 to 9 month project with a team size at no more than 10 people. Having no more than five iterations will then result in 1 to 3 use cases per iteration. The number of use cases for a project is an empirical observation from projects that have used use case driven development successfully.

Many projects have succeeded doing iterative and incremental development based on up to 50 use cases. But it seems like it is a question about granularity, and about how the use cases are count. If all extensions are described as separate use cases it can easily end up with 50 use cases. I only count the core use cases and not detailed extensions. This is closely related to GOALS DEFINE NUMBER.

But these numbers need more empirical evidence. A lot of parameters can influence the numbers. Many projects are for example dealing with adjusting existing functionality more than developing new functionality. Some projects involve new technology – others do not.

It will often be necessary to define a number of "pre-iteration" activities, especially for the first iteration. It is typically activities related to architecture and infrastructure. It can be done by defining a separate iteration to handle architectural issues, for example as an architectural prototype as recommended by RUP (Rational Unified Process) [11].

An example from Navision illustrates how a project successfully based their iteration plan on use cases:

> A development project was outsourced to a development partner. The project manager from the partner and the project manager from Navision met to plan the development based on use cases. The project manager from the partner should prepare by listing tasks. He has made a list, but hasn't related them to the use cases. We wrote the use case names on posters, and did the same with all the tasks. Each task was then placed below a use case, and what surprised us all was that it was possible to relate every task to a use case, even though they were identified independently of the use cases.
>
> Then the use cases were grouped into iterations. At first the project manager from the partner thought that there were too many dependencies to divide the development into several iterations, but we realized that a lot of these dependencies weren't a real problem. Maybe some things had to be implemented as stubs (for example a piece of code that simulates a non-implemented function) in the first iterations, but it could be done. The advantages outweighed the disadvantages. We planned five iterations, executed three, and the project was delivered on time.

SIZE THE ITERATIONS includes the principle of SPIRAL DEVELOPMENT where the use cases are "developed in an iterative, breadth-first manner, with each iteration progressively increasing the precision and accuracy of the use case set [1]."

Acknowledgements

Thanks to the workshop members at EuroPLoP 2003!

Thanks to Uwe Zdun for your careful and detailed shepherding of this version of the patterns.

Thanks to my colleagues Brian Jay Godkin, Dan Henriksen, Diana Velasco, and Susan Wiingaard for commenting on earlier versions of the patterns.

Thanks to all the Navision teams that have provided me with the stories and examples I'm using in the patterns – I hope you find that I have used the material in a decent manner.

Thanks to Neil Harrison for your excellent shepherding of the first version of the patterns for VikingPLoP 2002.

Thanks to all the nice people who workshopped the first version of the patterns at VikingPLoP 2002.

And thanks to Jim Coplien for inspiring me to begin this work!

References

[1] Steve Adolph, and Paul Bramble. *Patterns for Effective Use Cases*. Addison-Wesley 2002.

[2] Christopher Alexander. *A Pattern Language*. New York, Oxford University Press 1977.

[3] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison-Wesley 2001.

[4] Robert Biddle, James Noble, and Ewan Tempero. *Patterns for Essential Use Cases*. Technical Report CS-TR-01/02, April 2000.

[5] Alistair Cockburn. *Structuring Use Cases with Goals*. JOOP September and November 1997.

[6] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley 2001.

[7] Jim Coplien and Neil Harrison. *Organizational Patterns*. Org Patterns web site May 2003: http://www.easycomp.org/cgi-bin/OrgPatterns?BookOutline

[8] Neil Harrison. Harrison Patterns. Org Patterns web site May 2003: *http://www.easycomp.org/cgi-bin/OrgPatterns.book?HarrisonPatterns*

[9] Ivar Jacobson et al. *Object-Oriented Software Engineering*. Addison-Wesley 1992.

[10] Ivar & Sten Jacobson. *Use case Engineering: Unlocking the Power.* Object Magazine October 1996.

[11] Philippe Kruchten. *Rational Unified Process*. Addison-Wesley 1999.

[12] *Microsoft Solutions Framework*. MSF Resource Library web site May 2003: *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/tandp/innsol/msfrl/default.asp*