Software Development Best Practice #2 Use Cases

This second article of my Best Practices series is long overdue. However, the topic I promised to write about, Use Cases, is still as relevant now as it was then.

I assume that you are familiar with the basic concepts of Use Cases – if you are not, there are plenty of books, papers and websites where you can learn about them. I recommend Alistair Cockburn's or Ivar Jacobson's sites as good and reliable sources.

One problem with Use Cases is that there is no official standard for how to write a Use Case description. Even though the UML has a standard notation for Use Case diagrams, which show Use Cases and their associations with actors, there is no standard for describing the flow/dialogue within a Use Case and the other items you may want to include (pre-conditions, post-conditions, business rules, etc.). This means that different authors have developed their own template and style. As a result, it is hard to find two analysts who, left to their own devices and based on their experience, will completely agree on how Use Cases should be written. For that reason, software houses and IT departments should build a set of guidelines and ensure all their analysts adopt them. Variety can be the spice of life, but not where software specifications are concerned.

At RDF, we started with Use Cases in the year 2000, and have evolved a set of guidelines both for writing Use Case descriptions and for managing Use Cases within our projects. I will describe the key points.

Scoping and Estimating

At the project outset, Use Cases provide an effective "divide-and-conquer" technique for breaking down functional requirements. By focusing on actors and their goals, it becomes easier to establish system boundaries and functional scope. Initial effort estimates can also be produced by an outline analysis of the complexity and special needs of each Use Case.

I am not saying that Use Cases are the only way to analyse a system's requirements – business process models, domain models, state models and straightforward textual requirements, including non-functional, are also useful tools. However Use Cases are fundamental as they provide the unifying theme that runs through most, if not all, of the project work-streams.

During project inception we identify the key Use Cases and outline their purpose, interfaces and special requirements. Detail is added later, when the Use Case becomes due for development and test, on a just-in-time basis.

Planning

All our projects, if we can help it, have an iterative and incremental lifecycle – see my first best practice article. Use Cases are the units we use to plan the incremental development of the solution. If a Use Case is too large to fit within a single iteration, we break its development in two, or three, using whatever criteria are most appropriate in each case. For example, we can build the main flow in the first iteration, and all alternate flows in the next. Or we focus on dealing with a high-priority business case before we consider secondary cases. Ivar Jacobson, in a recent blog, uses the concept of Use Case **slice** to describe this.

Testing

Our test analysts plan and design their test cases and test scripts around the Use Case descriptions provided by the analysts. This is black-box testing. As a result, test analysts do not need to ask questions of the developers. The only area where a test analyst needs to look at the implementation is when it comes to preparing and checking database tables. It is normally straightforward to map from the business entities mentioned within the Use Case (see **Precision** below) to database tables and columns, but occasionally DBAs or database developers need to provide advice.

Changes

In our projects, system functionality is specified in two ways:

- Use Cases are used to capture interactive processes and batch processes.
- Business Services are used to capture functionality used by another system or by a Use Case.

No functionality exists in a system that is not captured either as a Use Case or a service; no business rule or visible aspect of the system exists that is not captured or referenced from a Use Case or service description.

Hence, when a new feature has to be added or a change is requested, impact on the system is analysed by considering how it affects existing Use Cases and services – this way nothing can be left out. Use Case and Service descriptions are living documents, they are updated as the system evolves and always provide an accurate and up-to-date picture of the system functionality.

Granularity

"Not too small, not too large" sums up our approach. Use Cases must support a single user goal and be potentially accomplished within a single user session. Anything that always requires multiple sessions, or time breaks waiting for business events, should be modelled as a business process, not a Use Case.

Sometimes all the functionality that is concerned with a single information area can be grouped within a single Use Case, provided it does not get too large. For example "Maintain Customer Information" may include adding a new customer, modifying customer details and removing a customer. It makes sense to treat this as a single Use Case, for practical reasons.

If a Use Case is inherently large and complex, we do our best to modularise it, using sub-flows and business rules, so it can be understood, designed and tested in slices.

Physically, each Use Case description has its own document file and is version controlled independently from the others.

Flows

We structure our flows using Main Flows, Alternate Flows and Exception Flows. The difference between alternate and exception flows is that the former achieve the desired Use Case goal, albeit via a different route, whereas the latter forces the user to abandon their goal. Think of the Use Case flow as a river flow. The goal is to reach the sea. Alternate flows are branches that end up in the sea, either by forming different rivers or by re-joining the main one. Exception flows are dead branches.

When a set of steps is used from more than one place, or in a loop, we describe it as a sub-flow. Sub-flows always return to the next step from where they are invoked, bar any exceptions.

We have also introduced the notion of Any-Time Flow, to capture an alternate flow that can be started from any step in the Use Case (think of pressing Cancel at any step during an application process).

User Interface

Traditional wisdom is that a Use Case description should not contain user interface details. For several years we have interpreted this rule to the letter and never put any information about screens or any other UI mechanism within a Use Case description. This approach has a number of drawbacks.

Business people, who need to review and approve Use Cases, find it hard to follow and maintain interest in a sequence of logical steps, even if written in a high-level business-oriented language. Screen visuals, prototypes and/or wireframes are far better at raising interest and getting feedback. Furthermore, if the logical flow and the UI spec are kept completely separate, it becomes harder to ensure that the implementation is consistent with both. After struggling with these issues for some time, we decided to add user interface specification artefacts to the Use Case description document, albeit in a different section. Rather than polluting the logical flows with UI details, we insert hyperlinks to the UI section. This way we can engage business people when reviewing Use Cases, while at the same time ensuring consistency between the logical flow and the UI view.

Purists would say that the dialogue in the Use Case should not be constrained by UI considerations. But we found that this is not true in most cases – the type of technology used for user interaction constrains the nature of the dialogue and shapes the logical flow of steps. The user experience, and hence the interaction, changes fundamentally, for example, from a web site to a mobile app – I would not advocate having a single “Book Flight” Use Case for both. Trying to abstract out the essential business steps in a single description is harmful, in our experience. If reuse of the business “essence” is the goal, this is obtained by use of Business Services and a business-oriented object model, not by making Use Cases abstract and generic.

Business Services

Use Cases are consumers of Business Services, just like external systems can be – we encourage our analysts to think in these terms right from the outset, rather than considering services as design artefacts.

Each Business Service has a specification model and a document that describes its functionality with no implementation bias. I have described this approach in a separate paper.

Precision

How can we keep Use Case and Business Service descriptions free of implementation detail but precise at the same time? We need precision because we don’t want to leave out any decision that affects the business outcome or that is visible at the system boundary. At the same time we don’t want to constrain how the designers/developers decide to architect the software solution within those boundaries.

We achieve this by describing behaviour in terms of manipulation of objects and their state. We develop a system object model while we elaborate the steps in each Use Case. The objects we use are not implementation objects – they don’t send or receive messages. We don’t use software classes in our model. We use ‘types’ of objects from the domain model. Each object type in our model represents one business entity that our system needs to know about, whether persistently or transiently, to provide the desired behaviour.

This ‘type model’ gives us a precise vocabulary to express business rules – we have a section of the Use Case for Business Rules, and each rule is referenced from the flow step(s) in which they are used. This provides traceability between behaviour specifications and data requirements, without going into premature database design.

I have described this approach in more detail elsewhere.

What about User Stories?

In recent years agile development methods such as Scrum have become widespread in the software industry. In most agile projects, functionality is not specified in Use Cases, but only captured informally as User Stories, which are then explored and elaborated by the whole team. This can save time and effort at the same time engaging and motivating the development team.

So does this mean Use Cases are no longer useful and should be disposed of in all projects? Are Use Cases incompatible with User Stories and agile methods? You may have guessed what my answer will be, but you will have to wait until I have time to write another instalment of my Best Practice series.

For comments or questions, email franco.civello@rdfgroup.com.