Designing Scenarios: Making the Case for a Use Case Framework

Reprinted from the Nov/Dec 1993 issue of *The Smalltalk Report*
Vol.3, No. 3
By: Rebecca J. Wirfs-Brock

Experienced object designers explore the design space from many different angles. They refine ideas of how their system should respond while they are in the middle of building and discarding ideas about how their design should work. Getting a design to gel involves making assumptions, seeing how they play out, changing one's mind or perspective slightly and re-iterating. Design is a difficult, involved task. It inherently is a non-linear process. Yet, we are asked to trace our design results back to system requirements. And, if we uncover some implications during design, we'd like to tune our system requirements to reflect necessary design compromises.

To meet these challenges, we need solid conceptual bridges to help us straddle the concerns of what the system must do (analysis) and how it will be accomplished (design). We also need techniques for adding detail, and driving out different perspectives during this process. In this column I'll describe experiences we have had bridging system requirements, object design and user interface design by applying use cases.

What is a Use Case?

Use cases, scenarios or scripts are roughly synonymous terms for important ways to focus our design activities. I prefer the term use case (although quickly saying it three times can leave your tongue tied) because it emphasizes usage.

A use case is a textual description of a sequence of interactions between an *actor* (roughly corresponding to an external agent or class of users) and the system we are designing. Use cases were first described by Ivar Jacobson in his book "Object Oriented Software Engineering A Use Case Driven Approach."

Use cases have been around in various forms for quite some time.

Jacobson, however, made the keen observation that use cases can be treated as refineable, extensible and even reusable specifications of system requirements. We've had these same goals for object designs. We know that it is harder to actually accomplish them than it is to talk about them.

Use cases are a pretty powerful modeling concept, once we know how to effectively build them. What sounds good in theory needs to be practically applied within a basic system development framework. A flock of questions come to mind:

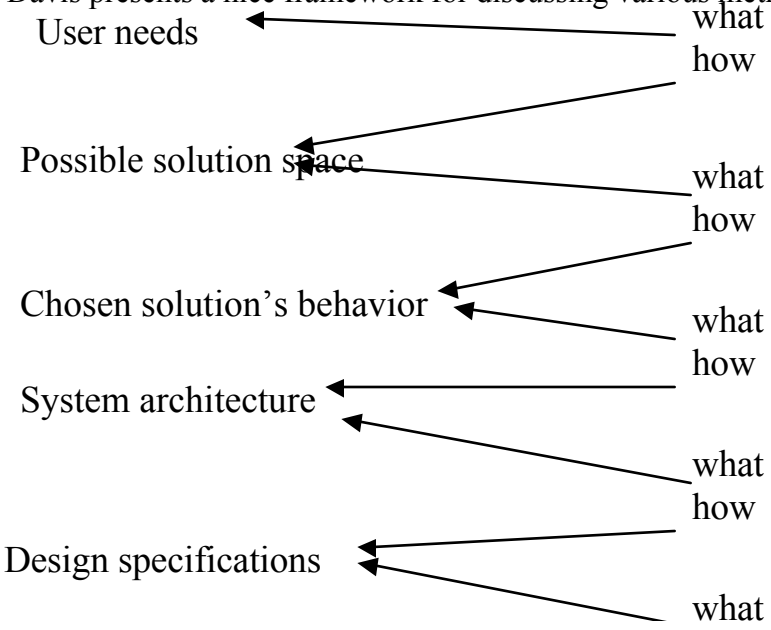
What process can you use to build good ones?
How should they be captured?
How detailed should they be? Are there different levels of detail?
When are you done finding and describing them?

I've had heated discussions about these exact same issues for object design. It isn't surprising that these themes keep recurring. People who build and describe software systems want to know how much they should describe before they truly understand what they are building. The answer to this question depends on how one intends to apply that descriptive information.

Many people claim to be using use cases. It's a trendy concept. Yet they all seem to be applying 'good use case construction techniques' at completely different levels of detail! This can be incredibly confusing to an innocent bystander, manager, student of design technique or end user!

Being the pragmatic, practical type, I really want to get to the heart of this matter. I've known for a long time that you really need to be aware of what perspective you are taking during a discussion.

I was stumped by the question, "What's a good use case?" until I read about the "what versus how" dilemma in Alan Davis' excellent book on software requirements [2]. Davis discusses the requirements analysis dilemma. Claiming that requirements are a statement of *what* not *how* is extremely simplistic (and insufficient for us to know how to pick and choose the level at which we want to be working). Many prominent requirements techniques are really modeling different requirements! Davis presents a nice framework for discussing various methods [Figure 1].



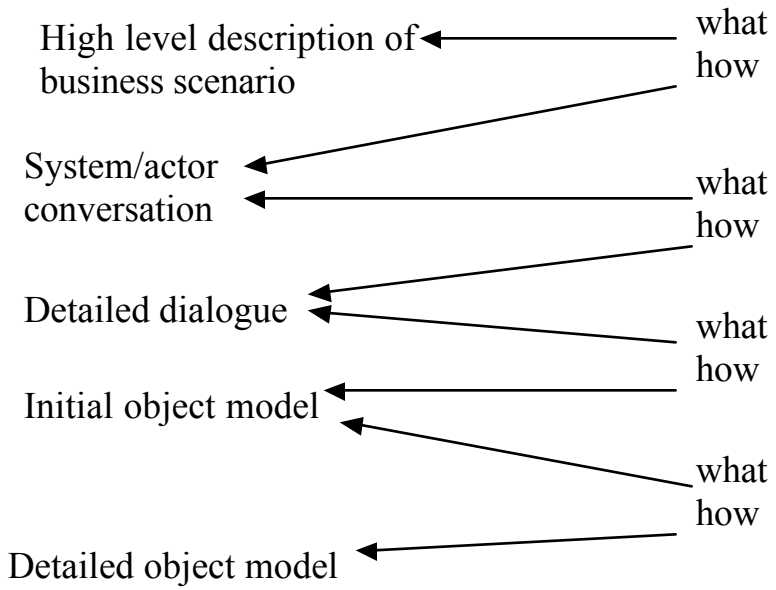
Each item in this figure can be said to rightfully be a requirement. From one perspective each item can be considered a "what" (a reasonable thing to include in a statement of requirements), or a "how" (something beyond scope). Depending on your viewpoint you can argue either for or against it. This is fun reading. It's good food for thought the next time you find yourself debating with your manager or colleague about whether some design activity is appropriate.

The key to solving our use case dilemma and keeping our sanity is to realize that one person's what is *always* another's how! Just as one size of requirements doesn't fit all, one canonical use case format won't work either. We need to formulate use cases slightly differently if we want to apply them to different purposes. Yet if we are careful not to get too arcane, these use case descriptions can be understood by a wide range of people.

What we really need is a conceptual map for describing and refining use cases, similar to one Davis proposes for thinking about requirements. Once we have this framework, I am perfectly content to initially state, "It depends" and try to ascertain real needs before coming up with a reasonable answer to the question "What makes a good use case?"

One answer doesn't have to fit all! New object designers need descriptions broken down into fairly detailed steps. Experienced designers are comfortable leaving out details that are easy to infer from the rest of the design. Those focused on designing user interactions need additional information. Teams transitioning formal system requirements on a large project into object designs undoubtedly need lots of precision.

Here's my proposal for a use case framework [Figure 2]. This framework could plug into Davis' picture after the system architecture and present an augmented view of his design specification models. We have experimented with several of these forms (with varying degrees of expressiveness and formalism) on several projects and in numerous mentoring and teaching situations.



A First Attempt at Describing a Use Case

Let's take a quick stab at writing a very high level scenario for our hypothetical Automated Teller Machine or ATM. We'll revisit and tune this use case to suit various needs.

> *Use Case: Performing an ATM Financial Transaction*
>
> > A bank customer can select a financial service from several available transactions. These transactions include cash withdraw or deposit, inquiring about an account balance, and transferring funds between two accounts. Once a customer has selected a financial service, she will be prompted to enter information necessary for performing the financial transaction. Upon completing a transaction, the bank customer may perform additional transactions, or indicate that she wishes to terminate the ATM session.

This description is so non-specific that we could present any design (a human teller might satisfy these requirements...or a fairly ridiculous design that had users entering their bank account numbers or cash amounts in Morse Code!) and argue that it met the requirements. Design students and developers need more guidance.

For large systems, there will be a wealth of additional requirements (ranging the gamut from user interface guidelines, detailed business function descriptions, banking regulations, to process specifications, and on and on). All this information still needs to be distilled into a comprehensible form in order to commence design. We can always refer to the wealth of supporting requirements material, we just don't want to be overwhelmed.

In less formal design efforts we need to supply more information. In either case, let's see how we might add more detail to this nearly content free description.

Our First Refinement

We have found it useful to clearly demarcate actor actions from system responses. This allows us to add more or less detail to either side of the conversation, as you'll see shortly. There are two central parts to this *system/actor conversational form:*

> a description of the actor's inputs to our system, and
>
> a corresponding description of our system's responses.

Together, these 'side-by-side' narratives capture a dialog between an actor and our system. There is also a list of alternatives to the main course of the use case. These alternatives represent

a reasonably complete list of conditions that system designers must be able to detect and to design appropriate responses for.

Here's our next cut at detail, an actor/system conversation:

Use Case: Performing a Withdrawal Transaction

Actor: Bank Customer

Overview: Bank customers can perform any number of financial transactions once they've presented the system with an unexpired, not known to be stolen bank card, and entered their valid personal identification number. The typical customer performs a single transaction before terminating a session with the ATM.

Actor Action	**System Response**
User indicates she wants to perform a cash withdrawal	
	Present the user with a list of accounts
Selects a particular account	
	Prompt the user for cash amount
Indicates cash amount	
	Validate available funds on hand
	User cash amount must be in multiples of available denominations
	Update account balance
	Withdraw request must be within daily ATM limits and cash in account
	Log transaction on external record and prepare receipt information
	Dispense cash and sense when user has removed it from dispenser

User retrieves cash

Ask user if another transaction is desired

User indicates she is finished

Print out receipt and eject it

Eject user's card

Alternatives:

1. Insufficient funds on hand.

2. Insufficient funds in customer account.

3. User doesn't respond to any part of the dialog (within a sufficient time period).

4. User wishes to perform an additional financial transaction.

Use cases should be constructed by business domain knowledgeable people. The key to building a good use case is to remember that it serves two purposes: it will guide developers and be reviewed with clients. Use cases should be written for *both* audiences. Actor interactions and system responses need to be described at a fairly high level.

On the other hand, descriptions of system responses must contain sufficient detail so that object designers working with analysts can design a reasonably detailed object model. Walking this fine line between sufficient detail and bogging down in details requires practice and critique.

Level of Detail

The sample use case we wrote still needs more detailed descriptions before we can design our system. In particular, the system responses need to be expanded upon to include:

Steps and logical sequencing of actions that must be performed by the system.

A description of necessary information that must be supplied by the actor. Reasonable defaults (if any) for information not supplied.

Description of any data validation or business constraints that must be checked before performing a system action.

Format of reports that are generated.

Timing of and contents of any *significant* system feedback.

This could be captured in other documents, or less formally, much of this information might be directly placed in our object model. It needn't be crammed into either side of the conversation. Side notes and additional constraints make it hard to follow the conversation thread.

We wrote our system/actor conversation for a concrete situation, withdrawing cash. We could have written it more abstractly, where the system response and actor conversations would describe any of the permissible transactions. If we wrote at this more abstract level, we'd have to remove a fair amount of detail from both the actor and system dialogs. For example, since not all transactions involve amounts specified by the user, we'd have to state that the "user is prompted for additional information, if required," and that the "bank customer enters information." We would also have to remove alternatives that don't apply.

This prematurely is too abstract for my tastes. This feels uncannily like the process I go through when I refactor responsibilities during detailed design. I do this *only* after I have responsibilities assigned to concrete classes. When I am carving up the system's responsibilities and finding more general ways of stating things, I also take pains to assign details to concrete classes. I'm not losing class specific behavior during this refactoring.

I find it useful to keep use case conversations pretty specific for another important reason. They are understood by analysts, users, and developers. We can periodically review and verify correctness with clients. They will be refined over time to reflect actual implementation and to include more detail, particularly on the system response part.

Tuning the Conversation

Conversations can also be worked on in order to tune the user interface of our application even before building a prototype. The most likely thread through the conversation is termed the *main course.* Other optional paths are alternatives. On a number of projects, we have focused a lot of attention on designing the particulars of conversations. This fine tuning really pays off for systems where improving the quality of human-computer interactions is a high priority.

In these situations we pay particular attention to finding the main course and painstakingly ensure that this indeed is the most common task that users want to perform. We also use a variety of user interface design techniques to construct and test theories about preferred conversation patterns. This is a lot of work. Conversations can be crafted by interface specialists (and iteratively prototyped by designers) if the project's size and scope warrants this attention, or they can be done more informally.

Even though it is a good idea to separate the details of user interface and information presentation from our business objects, the way a user converses with our software can have significant impact. In my experience, no application area is immune. I personally know about oscilloscope control software, customer information systems and trip planning applications where seemingly

minor interface features placed significant demands on the underlying object model. Any effort to work out the interface issues early made our job of developing the other parts of the software that much easier.

Conclusions

Users can work with analysts *and* object designers to formulate and tune system requirements. People from business, analytical and object design disciplines can come together, learn from each other and generate meaningful descriptions of systems that are to be built. Each participant and each project has slightly different concerns and needs. Practical application of use cases can go a long way to improve our ability to deliver just what the customer ordered.

References

[1] Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard, "Object Oriented Software Engineering a Use Case Driven Approach," Addison-Wesley, 1991.

[2] Alan Davis, "Software Requirements Objects, Functions & States," Prentice Hall, 1993.