Developing System Requirements

Table of Contents

- Watch: Writing Good Requirements
- Tool: Functional Requirements
- Activity: Make Your Requirements Functional
- Discussion: Viewing Requirements From an Outside Perspective
- Watch: Following the Requirements Rules
- Quiz: Bad vs Good Requirements
- Watch: Setting Up Requirements Tables
- Tool: Requirements Table Template
- Watch: Balancing the Spirit of the Rules with the Letter
- Activity: Use a Constant to Resolve a Requirements Issue
- Watch: Recording Your Constants' Definitions
- Tool: Requirement Constants Definition Table Template
- Watch: Delving Into Your Requirements
- Read: Tips For Delving Into Requirements
- Activity: Identify Missed Functionality
- Activity: Rewrite Problematic Requirements
- Discussion: Fixing Requirements
- Read: Creating UCBDs for Your Other Use Cases
- Watch: Populating Activity Diagrams
- Tool: SysML Activity Diagram Template
- Assignment: Course Project, Part Two—Improve Your Requirements and Create an Activity Diagram
- Module Wrap-up: Requirements That Best Serve Your Project
- CESYS522 Glossary
- Tool: Course Tools

Welcome to Your Course

Video Transcript

So you define the scope of your project--you've got your context set up that it's going to operate in, you've got a complete set of use cases that must be met to meet your customer's needs. But now, how should your system interact with all the elements in that context you identified? What does your system need to do to successfully complete each use case?

That's what we're going to explore in this course. We're going to use a tool called a Use Case Behavioral Diagram to define professional, functional requirements that describe what any valid solution must do throughout your use cases. We're going to do so in a way that is both professional to meet your customers needs, but not too prescriptive to allow your team to use all of their talents towards making your project a success.

Course Description

You have identified the scope of your system. You know what your system is responsible for and what your system must interact with in order to be a success. Now you need to discover and flesh out the responsibilities of your system, and capture these responsibilities as formal functional requirements.

In this course you will build a Use Case Behavioral Diagram (UCBD) for a specific use case. The UCBD will help you articulate all the functions your system must be able to achieve successfully throughout that use case. You will also use this tool to develop an understanding of the difference between functional and structural requirements. With a clear understanding of requirements, you will have defined what any valid solutions must be able to achieve throughout your use cases. You will also convey the information found in a Use Case Behavioral Diagram using a SysML Activity Diagram. Experience with this alternative format will expand your options for communicating use cases' requirements with your team.

What you'll do

- Determine all the functions your system must be able to achieve throughout each use case
- Develop Use Case Behavioral Diagrams to define formal, verifiable requirements for a system

- Use an Activity Diagram to create a SysML variation of a Use Case Behavioral Diagram
- Develop an understanding of the difference between functional and structural requirements

Faculty Author

David R. Schneider

Director of MEng Studies in Systems Engineering

Cornell Engineering

Cornell University

David R. Schneider earned his Masters and PhD from Cornell University in Mechanical Engineering with a concentration in Controls & Dynamics in 2007. David has taught at both Cornell and Columbia University. David Schneider's research has traditionally focused on the realm of NP-Hard Computer Science Problems and Controls for Robotic Systems in centralized and decentralized as well as autonomous and semi-autonomous systems. His most prominent research is his creation of the G*TA (G-Star-T-A) task allocation algorithm and his work as Program Manager of the Cornell RoboFlag program, with notable applications including AFRL UAV controls and NASA/NOAA unmanned boat designs.

With a strong focus on education, David's endeavors have included the creation of the Cornell Cup, Innovative Embedded Design National Competition; leading Cornell University Sustainable Design (CUSD); and the broader impacts video game creation for the NSF Expeditions in Computing Grant on Computational Sustainability. David has led the efforts to make Cornell the first university to officially partner with Make: and is a leader in the Higher Education Maker Alliance working with the White House Office of Science and Technology Policy.

[Back to Table of Contents]

Read: Preview Your Course Project

Throughout this course, you will practice applying the teachings of each module in a multi-part course project. You can review all relevant information on the **Grades** page of this course.

Since each module builds on the previous ones, it is essential that you work through the course in the order it appears. At the end of the course, you will submit your completed project for facilitator review and grading.

☆ Key Points

You will apply the content from this course in a multi-part project.

You must submit all prior assignments to access the final project part.

Course Project Parts

Preview the course project below. As you work through the course, reflect on how this relates to the course content and to your experience.

Note: Though your work will only be seen by eCornell, you should take care to obscure any information you feel might be of a sensitive or confidential nature.

— Course Project, Part One — Create a Use Case Behavioral Diagram

In this part of the course project you will use the UCBD Template to create a Use Case Behavioral Diagram for a system you want to develop. In your UCBD you will determine the main actors, the starting and ending conditions, and what your system should do, stated as functional requirements.

— Course Project, Part Two — Improve Your Requirements and Create an Activity Diagram

For this portion of the project you will revisit the Use Case Behavioral Diagram from Part One of the project and delve into your requirements.

You will examine which of your statements are structural versus which are functional, and then you will rewrite your structural statements so that they are formal, verifiable statements. With your completed UCBD you will create a SysML

variation of an Activity Diagram and a corresponding Requirements Table for that use case.

Back to Table of Contents

Module Introduction: Developing Use Case Behavioral Diagrams (UCBD)

You have a great system in the works. But in order to adequately build this system you must determine all functions your system must be able to achieve throughout each use case. To help you discover and flesh out the responsibilities of your system, you need a Use Case Behavioral Diagram (UCBD).

In this module you will develop a Use Case Behavioral Diagram that will help you formalize the relationships between operator, system, and other elements in your system. Your UCBD will also help you effectively communicate the requirements of your system to others.

Back to Table of Contents

Read: Defining Your System

Use cases present scenarios or describe ways a system can be used. You need to create functional requirements to define what your system must accomplish during these use cases.

By formalizing these functional requirements, you create a technical definition of what any valid solution to your problem must do. You also take a critical step toward enabling every designer to understand what your system must accomplish. At the same time, functional requirements allow for a solution that leverages the team's talents and creativity.

☆ Key Points

The parameters of your system are based on your stakeholder's perspectives, the context in which your system must operate, and the use case scenarios that your system must accomplish.

Functional requirements allow designers to understand what a system must accomplish, while still allowing room for creative and valuable solutions to meet those requirements.

The **Defining Your System Requirements** guide describes a step-by-step sequence for analyzing use cases and formally determining functional requirements. This course explores the steps found in the guide; however, the guide goes into further detail and includes a brief consideration of performance measures, which is outside the scope of this course. Keep this guide handy as a reference.

As you take these defining steps you will have a far greater understanding of what it is you must achieve. You will be able to create a far more successful, complete, and efficient solution than you would have been able to without these steps toward defining your system.

Back to Table of Contents

Tool: Defining Your System Requirements Checklist

Download the Tool

Defining Your System Requirements Checklist

You want to be able to clearly communicate the functions of your system. In order to do this, you will need to capture specific functional requirements. The **Defining Your System Requirements Checklist** guides you through the process of developing and capturing requirements that describe your system. Keep this checklist handy and use it whenever you need to work through the requirements process. Once you have completed these steps, you will have well-defined requirements for your system.

Note: The checklist steps have been taken from the more comprehensive **Defining Your System Requirements** guide; however the steps numbered in the checklist don't necessarily align with the guide's numbered steps. Still, all the necessary steps are provided within the checklist.

Back to Table of Contents

Watch: Describing Your System Functionally

A Use Case Behavioral Diagram is used to define the function performed in a single use case. Think in terms of what the system will be able to do, what the use case requires of the system, or what the system is designed to achieve for a specific function. Your Use Case Behavioral Diagram will provide clear instructions on how your system will solve the overall challenge.

In this video Professor Schneider provides an introduction to the Use Case Behavioral Diagram and describes the preparation needed before you begin building your diagram.

Video Transcript

With your use cases defined, now we're ready to define the functions that not just your system must perform, but any valid system must perform throughout your use cases. Our main tool for doing this is going to be the Use Case Behavioral Diagram. Now don't get this confused with the use case diagram, which shows the relationship between different use cases.

A Use Case Behavioral Diagram is a way of defining for a single use case the clear beginning and ending conditions, as well as what are all the key elements that are involved, which, of course, is going to be your system, as well as your operator, which for now we'll refer to it just as the main outside source of action or driving the stimuli for action to occur in that use case. Also going to include as elements, what are other pieces of equipment, other sources of information that you might have. Basically, anything that your system interacts with throughout that use case. And then through a connected series of statements we're going to find what your system and all the other elements must do throughout that use case.

Now, at face value this sounds pretty simple, but the key to doing this well is actually one of the most fundamental system engineering skills, which is to be able to describe things functionally. And let me give you a few examples. When we say a car is going to drive, you could say that that's something a car does. Well, it's actually meaning the function of providing transportation. So you don't say, "a car drives," you say, "a car performed a transportation function." Similarly, a refrigerator doesn't cool food, it maintains an environment that prevents food spoilage. That's the function it provides. Similarly, a medical treatment doesn't produce a specific chemical reaction so much as it provides a function of reducing symptoms, bettering health conditions, or reducing infections. It's that functionality that we want to get at the heart of.

And understanding what the functions that must be accomplished is where we need to keep our focus on before we think about how we're going specifically meet those functions. Said another way, we don't want to think about what is the specific solution that we're going to use to meet this need before we understand the entire problem. This is especially important on teams because no one person, regardless of how talented they are, can make the best design decisions at every aspect of the project. So we need to be able to find the functional needs that they must be able to achieve throughout the design process, and then allow the very talents of the entire team to determine what solutions can work best for all the needs.

If you don't understand all the needs first, we might wind up designing something that's inefficient, or even invalid, that only means, perhaps, a piece of the problem needs, that has, perhaps, not even the best solution for the overall problem that you're trying to deal with. And this is a situation that you may see happen many times, where you might imagine someone designing something, or a whole team of people designs it. "Wow, we feel like we have created the best product that we can ever come up with." Then we wonder, why isn't it selling? Why aren't they buying this? You may be very talented, but you solved the wrong problem because you didn't understand all of the needs.

And being such an important tool, a Use Case Behavioral Diagram, of helping to pull out what are these key functions. There's actually several different forms that we could follow in doing this. We'll be going over an official SysML form much later, but let's start with a widely used spreadsheet form, which is much easier to work with.

Back to Table of Contents

Activity: Identify Your High Priority Use Cases

In practice, you will create Use Case Behavioral Diagrams (UCBD) for each of your use cases. In this course, however, you'll focus on just one use case to get some practice.

At this point you should have a healthy number of use cases (at least 8) that you've already diagrammed. You'll want to choose a fairly important use case from among these for your UCBD practice. You do not need to choose a use case just yet, but in this activity you will be creating a short list of your more important use cases.

Note: As you work through the activities in this course you will be making progress toward completion of your Course Project. Although you won't turn in anything after completing an activity, you will need to save copies of your documentation produced in the activity. Some or all of it may be necessary to fulfill the course project requirements.

Prioritize Use CasesPrioritize Use Cases
This is Complete When...This is Complete When...
Back to Table of Contents

Tool: Use Case Behavioral Diagram Template

Download the Tool

Use Case Behavioral Diagram Template

You need to discover and flesh out the responsibilities of your systems and capture these responsibilities in a professional way. In order to do this you need to use a Use Case Behavioral Diagram (UCBD). This tool will help you communicate the functions of your system for a specific use case. This tool will also help you show the relationship and interaction of your system with its operator and other elements external to the system.

This is a template for your Use Case Behavioral Diagram. Use this template as a starting point any time you build out a diagram for a specific use case.

Note: This tool is a Microsoft Excel workbook. One sheet includes the template you can use as a resource to create your own UCBD. Another sheet shows a completed sample that you will see developing in stages as you watch the videos. A third sheet has a concise list of guidelines that will help you write correct functional requirements for your system.

Back to Table of Contents

Watch: Setting Up Swimlane Columns

The Use Case Behavioral Diagram is set up in columns, or swimlanes, to help manage the information contained in a single use case. In the template each main element that interacts with the system, as well as the system itself, is listed as a header.

In this video Professor Schneider describes the different columns and provides an example of how each element plays into the use case of your system.

Video Transcript

There's a couple places where we could begin in developing our Use Case Behavioral Diagram, but let's start with our actual main elements that are involved in this use case. Elements are listed as column headers in the template, and these columns are also often referred to as swimlanes. The first, or leftmost, column is traditionally referred to as the operator column, or the Primary Actor column. Both operator and primary actor mean, essentially, the same thing. This is the column that drives the main action. It's the main stimuli that actually causes this use case to occur.

Now let me give you an example. So let's imagine, I'm thinking of building a toy catapult. And maybe an example use case I might have is child loads catapult, or better yet, I have child loads system. And this is better, of course, because now I'm thinking more functionality, right. I'm not saying that I'm designing a catapult because later on I might find that there's another launch system that works even better than a catapult towards me and the set of functionality that I'm going to discover here overall. So, but regardless of your situation, child loads catapult or child loads system, the operator is the child because they are the one that's driving the action for loading this catapult-like system overall.

Now, the column immediately to the right of the operator column is reserved for your system, and is aptly named The System. Now, do note that not all use cases have an exterior operator. So a use case name that is simply System Does X might have as your leftmost column simply, The System column, implying that your system is driving all the action instead of some kind of exterior operator.

After you have the system column, you can add as many additional columns as you like for anything else that's external to your system that is involved in this use case. For example, you might have other things that you might want to launch in your catapult-like system that are not actually a part of your actual catapult toy. Or said another way, these are things that you

might want to launch that aren't actually included in that box that you're selling your toy catapult in, overall. So again, it could be other things that you might want to launch.

Now, there don't have to be any additional columns in your case behavior diagram, but you add as many of these columns as you think you might need for any of those external elements, overall. But once you've added one column and only one column for your system, a separate column and a separate column for each key exterior element involved in the use case, you're ready to move on. You can always add or remove ones later on, and quite often use case behavior diagrams are a very iterative process.

But in doing those iterations, I want to offer one very, very important safety tip. Do not, do not split up the system column into multiple columns. This is very important for several reasons. First of all, it's the same reason why we've referred to our catapult system as simply the system and not the catapult. Splitting up your system column into multiple different columns, or parts, or subsystems, forces any final solution to also be considered only as being split up in that way. And again, in our process of going through this Use Case Behavioral Diagram, we may find out that the functionality we're trying to achieve can actually be achieved in a much different way than we initially thought at the beginning. So, by adding all the different additional subsystems or parts, or splitting up that system, we're basically narrowing a solution space way too quickly. So resist the urge.

Not only for that purpose, but also for the purpose that many government documents will actually only be accepted if you have only one system column. Just one system column, overall. So again, very important to have only one system column. Now, with that being said, if you are working on a larger system, overall, and you have control over just one subsystem of that overall system, you may treat that one subsystem as your single system column and then represent all other subsystems as exterior columns. Again, this is okay because you're treating your subsystem as the one thing you have control over, and all those other subsystems as exterior things because those, again, are elements that you don't have direct control over.

Back to Table of Contents

Activity: Set Up Your UCBD

In this activity you will start to build a Use Case Behavioral Diagram for one of your high priority use cases.

Task: Set Up Your UCBDTask: Set Up Your UCBD
This is Complete When...This is Complete When...
Worked ExampleWorked Example

Back to Table of Contents

Download the Tool

Use Case Behavioral Diagram Template

Watch: Creating Additional Columns and UCBDs

In creating your UCBD, you may discover a need for additional columns to further explain additional elements. You also may discover a use case very similar to your original use case, but with some minor variations. When you discover a new use case, the solution is to give it its own separate UCBD.

In this video Professor Schneider uses the catapult toy example to show ways an additional column can further clarify a use case. He also shows where to include notes in your diagram. In addition, he explains when to create a new UCBD to further explore a variation to a use case.

Video Transcript

In creating Use Case Behavioral Diagrams, there's often this kind of anxiety that builds up, that feels like, I only have so much time to get this done and hence, I've got to get it right the first time that I do it. But let me tell you, the energy that you waste in dealing with this anxiety and worrying about whether it's perfect the first time or not typically will cost you more time and energy than if you just took a good stab at it, then allowed yourself to go back over different parts of it. Remember, a Use Case Behavioral Diagram is a discovery process, and no great explorer ever maps out the perfect course before setting sail.

So, regardless though, let's address a few common issues and questions to help you get off to a good start. Now, in the last example that we talked about, in the catapult example system, we had a separate projectile column. Now, you may say, why do we do this? Why do we have a separate projectile column? Now, if your projectile comes as a part of your system, and is special to your system, and you have full control over its design, then you wouldn't add a separate column for it. But if any one of those above statements is not true, you may likely need to as separate column.

As an example, perhaps your toy has to work with Nerf darts as an example standard projectile used in many similar toys. That Nerf dart was probably designed way before your toy was even conceived. And since you now have no control over that projectile's design as an already established standard, it makes sense to add as a separate column. Because again, even though it might be included as something in the box with your toy, you don't actually have control over its design. So you treat it as if it's an exterior column in that situation.

Now, what if you designed your own projectile that comes with your toys, so you've got your own special projectile for that, but you expect that the user will shoot things other than that projectile with their toy? That's another good example where you might have a separate column for that projectile. However, trying to think about all the different kinds of projectiles in the same exact way might be too much to ask in a single Use Case Behavioral Diagram, so you also need to be able to present limits to the person who is reading this on what the projectile might be for this particular use case.

The way that we communicate this is through notes at the bottom of the Use Case Behavioral Diagram. For example, a good note might be, "We assume that the projectile is not a living thing, like a pet gerbil," as an example. So that would be a note that you would put at the bottom of the Use Case Behavioral Diagram. Lots of times these notes may also help you recognize you need additional use cases, such as child load system with living creature, and then you can handle those situations separately. But again, you put the notes there in order to help give some additional information and say what is the scope of this use case, overall.

And to this end, notes are very, very important, because without them someone else, or even yourself, might not realize weeks, or even years, later when you're reviewing this document, what you actually meant when you first wrote it. So add more notes, add more notes, add more notes. Add as many notes as you think could be useful to somebody else or, again, yourself later down the line.

Let me give you one other common example that you might run into before we talk about our Use Case Behavioral Diagram so far. Let's assume that there might be potentially a parent who might be playing with our child because that's a common situation you might have. And so you might ask that the situation, do I need to have a separate use case with the parent, or can I include the parent in the same use case that I'm describing right now in the Use Case Behavioral Diagram?

The answer to this question is basically, if your system has to do different functions depending upon who is present or whether other things are present, in this case whether or not the parent is present, it be whether or not a piece of equipment, or the environment is different, etc., if your system has to do different things depending upon different exterior conditions, then you should have different use cases for them. So, for example, where the parent is not involved, child load system, and one that does involve the parent, child load system with parent. And then create a separate Use Case Behavioral Diagram for each one of those.

Personally, the way I like to try to tackle this problem is I usually try to complete the diagram with the most basic version first. So I probably would just try child load system as my initial use case that I would start out with. And then review it later with the idea of a parent also being present to see if that changes what my system would have to do. For example, some toys actually come with special features that are meant to attract parents to buy the toy, that would come out of having a parent present as compared to just having the child use them.

This kind of iteration is very natural, and changing new use cases as you develop Use Case Behavioral Diagrams is, hence, also very natural and perfectly acceptable as well. It's also encouraged that you quite often have longer use case names in order to help to distinguish between the various different variations that you may be dealing with.

Back to Table of Contents

Activity: Populate Your Swimlane Column Headings

In this activity you will continue building your Use Case Behavioral Diagram by determining the main actors in your use case.

Task: Populate Your Swimlane Column HeadingsPopulate Your Swimlane Column Headings
This is Complete When...This is Complete When...
Worked ExampleWorked Example
Back to Table of Contents

Activity: Adding Notes to Your UCBD

In this activity you will add notes to clarify the information within the rest of your Use Case Behavioral Diagram.

Task: Add NotesTask: Add Notes
This is Complete When...This is Complete When...
Worked ExampleWorked Example
Back to Table of Contents

Watch: Revisiting Your Use Case Diagrams (UCDs)

As you move through the process of developing your system, you should view all your design documentation as a work in progress. Your understanding of your system will probably change as you uncover more details about it. These discoveries may lead you back to the use case diagrams (UCDs) that you developed at an earlier time. In some cases, you may wish to add use cases or make modifications to these UCDs.

In this video Professor Schneider provides an example of how UCBD development leads to modification of a use case diagram (UCD).

Video Transcript

When creating a Use Case Behavioral Diagram it's very natural to iterate on your use cases themselves. For example, if we had child load system, we may discover in the process of developing its Use Case Behavior Diagram that we want to have another use case called child load system with parents. Now, these two use case variations might show up in a couple of different ways in your associated use case diagram.

One might be that you simply include another use case with the parent named directly in the name. Perhaps you might also include the parent as a secondary actor connected here with a dashed line. You may also choose to show a connection between these two use cases. Here we show that you can use a generalization arrow, which shows that the parent version must do everything that the non-parent version must do, but in a special way.

Your company or your customer may have a preference which way they prefer, but the important thing is that you remain consistent in the way that you handle these things throughout all of your team's diagrams. Personally, the generalization arrow is the way that I like best.

Back to Table of Contents

Watch: Setting Beginning and Ending Conditions

Your system may require specific parameters or a certain environment before it can be used. These initial conditions should be listed in your diagram. The same goes for any conditions that specify the completion of your system's function. These should be listed as end conditions.

In this video Professor Schneider describes what information should be included in your beginning and ending conditions, and where these items should be placed in your UCBD.

Video Transcript

Now let's add beginning and ending conditions to our Use Case Behavioral Diagram. The beginning and ending conditions are what must be true about your system at the start and at the end of your use case. They often indicate what are any major inputs or outputs to your system over the entire use case. And these can be either physical inputs or outputs or informational inputs and outputs, as well.

In your beginning and ending conditions you might also list the condition of anything that your system might be operating with or detecting. For example, any of your swimlane columns, such as your operator or other pieces of equipment. You can also list the condition of things that are not swimlane columns. Perhaps a common example of this is an expected amount of a resource. Maybe at the beginning you expect something to be at least half full. And then by the end of that use case you expect it to be all used up, whatever that resource is. So it's important to list those elements in your beginning and ending conditions as well.

Now, it's also important to list any assumptions you make, and you have to think about this very creatively and completely. For example, maybe you're designing something that you expect to be used in the kitchen. Well, what if somebody starts to use this when they're cooking outdoors at a campsite? And you start to think, well, wait a second, that's not the way I was thinking about someone using this, so I don't need to worry about it, right? No, you absolutely need to worry about that. If it caused your system to have to do something differently, it could cause your system to have to be designed very differently. Therefore, you must address it.

So indicate the assumed environment or any other assumptions you have for your use case in the starting condition. And then create a separate use case for any other variations of that assumption. So any other environments that you might expect it to be used in. Thinking

about these conditions can also help you make decisions on your scope. You might decide, I know I haven't thought about that before, so now I need to include it in my scope, this is intended for indoor use only or for use under X conditions only as a part of that as well. And make sure you include that in those starting or beginning or ending conditions, or at least in the notes of the Use Case Behavioral Diagram.

Because one of the most important things that beginning and ending conditions serve is actually as a communication tool, overall. And one of the things that you may find is as you are developing this is that you don't even know everything that you want to communicate quite yet about it. You don't know what your system is yet. How are you expected to know what state it might be in? That's not uncommon. You're going to have to go back and update these Use Case Behavioral Diagrams later. But for right now, take your best guess, listing assumptions that you make and information about those assumptions in your notes so you can come back and update these things effectively as you design your system better later on. And then this document not only serves as a design document, but also as a final documentation document.

Common assumptions that you may want to start out listing right away are, "the system is turned off," or "the system has already gone through initialization." Maybe "an emergency condition has been detected" or "emergency condition is occurring but has not yet been detected" as a part of this. So make sure you include all those different kinds of elements in your beginning and ending conditions that you have. Because again, these are really essential for communication because you don't want to run into the situation where, "oh, I thought we were talking about this when we were talking about this use case. I assumed we had X," or "wait, wait a second," another teammate says, "I thought we were talking about this with this use case." Uh oh. We better start making some adjustments here.

You want to make sure that you check over your Use Case Behavioral Diagrams with your immediate team before going forward. And usually the beginning and ending conditions are a great place to start with before you start to flesh out the rest of your Use Case Behavioral Diagram with your team. So, in short though, you should make sure that your beginning and ending conditions clearly state how your use case starts and how your use case ends. And your overall team is in agreeance with it.

Back to Table of Contents

Activity: Determine Starting Conditions

In this activity you will determine the starting conditions for your use case and record them in your Use Case Behavioral Diagram.

Task: Determine Starting ConditionsTask: Determine Starting Conditions
This is Complete When...This is Complete When...
Worked ExampleWorked Example
Back to Table of Contents

Activity: Determine Ending Conditions

In this activity you will determine the ending conditions for your use case and record them in your Use Case Behavioral Diagram.

Task: Determine Ending ConditionsTask: Determine Ending Conditions
This is Complete When...This is Complete When...
Worked ExampleWorked Example
Back to Table of Contents

Watch: Completing Your UCBD

Once you've established the components of your UCBD, you need to explore what functions occur between the beginning and ending conditions. This sequence of events makes up the body of the diagram. The UCBD needs to capture all use case behaviors of your system using formal functional requirements. But what makes a requirement formal, and what does it mean to be functional?

In this video Professor Schneider describes the basic rules for populating the body of your diagram. In particular, he introduces formal functional requirements for system behaviors.

Video Transcript

Now we're going to determine the sequence of events and functions that must occur between the beginning and ending conditions in our Use Case Behavioral Diagram. This is often referred to as filling out the Use Case Behavioral Diagram's body. Now, the key to this is going to be to determine and formalize functional requirements that your system is required to do throughout this use case. We'll be spending a lot of time going over functional requirements, but let's first go over some basics of formatting first.

Now, use cases often begin with some kind of trigger action, often initiated by the operator, as shown in the example. Here the first action is the operator pushing the toy's projectile receptacle into position. In some variations of Use Case Behavioral Diagrams the trigger action is actually included as a separate part of the beginning conditions. Sometimes a trigger is listed as a completion of another use case. In some use cases though, there isn't a distinct trigger, so don't worry if you can't come up with one, or even if you start in a column that's other than your primary actor or operator column that you have. That's okay.

The key thing is that once you have written your very first statement for how your use case begins, you simply go to the next row and then write what happens next. In this example we say, "the system shall detect the receptacle in the proper position," and "the system shall secure the receptacle in the proper position." That's a good start, but notice actions done by anything other than your system is typically written in a slightly less formal way, simply stating what is occurring. What is written in the system's column, however, is written in a far more formal definitive statement. "The system shall do something." We are writing formal requirements now.

And we're going to be spending a lot of time on how to do these well, but there's a big difference in the value of a Use Case Behavioral Diagram and the way that we handle this. Particularly distinct is whether we're just going through the motions of something, or whether we're actually creating good functional requirements. And good functional requirements are often considered the most valuable product of doing a good Use Case Behavioral Diagram. Again, let's first finish the basic rules of a diagram for now.

Now, actors, actions, and formal requirements, referred to generically as statements, should be written in a chronological order from beginning to the end condition. Each statement should be written in its own separate row. No row should have more than one statement in that row, even if things are basically happening at the same time. Each statement must be written in the swimlane corresponding to the element, with the primary element responsible for that statement. If more than one entity is responsible, try splitting the statement into more than one statement.

For example, if an element is responsible for providing input or having some other kind of exchange between a couple of different elements, a statement should be placed in the element column providing the input. Then in the next row another statement should be written in the swimlane of the element receiving the input, indicating how the input is handled. It's better to have more statements than unclear assignment of responsibility, or a single statement that tries to capture too much of what's going on.

Another common situation you'll find in Use Case Behavioral Diagram development is that, as you're going through the body, you find that, well, if this one particular case would happen, the logic flow of events would happen this way. Otherwise, we'd have a different series of events that occur. The way to handle this is you actually have a different Use Case Behavioral Diagram for each logic flow. Said another way, each Use Case Behavioral Diagram should only follow one logic flow, and you have as many Use Case Behavioral Diagrams as necessary in order to handle different logic flows.

Now, those are the basic rules, and you can now create a Use Case Behavioral Diagram and get something out of it. But we're still a far cry from doing this well, and getting the real benefit of this tool.

Back to Table of Contents

Activity: Determine Your System's Requirements

In this activity you will determine the requirements necessary for your use case to proceed from its starting condition to its ending condition. You will record these requirements in your Use Case Behavioral Diagram.

For now, focus on the UCBD you have been working on. Later, you can create additional Use Case Behavioral Diagrams as needed.

Task: Determine RequirementsTask: Determine Requirements
This is Complete When...This is Complete When...
Worked ExampleWorked Example
Back to Table of Contents

Assignment: Course Project, Part One—Create a Use Case Behavioral Diagram

In this part of the course project you will use the **UCBD Template** to create a Use Case Behavioral Diagram for a system you want to develop. In your UCBD you will determine the main actors, the starting and ending conditions, and what your system should *do*, stated as functional requirements.

In Part Two of the project you will delve further into the functional requirements for this use case of your system.

Completion of all parts of this project is a course requirement.

Instructions:

1. Download the **course project document**.
2. Complete Part One.
3. Save your work.
4. You will submit your completed project at the end of the course for grading and credit.

Do not hesitate to contact your instructor if you have any questions about the project. You will add to this document as the course proceeds and will submit it to the course instructor at the end of the course.

Before you begin:

Before starting your work, please review the **rubric** (a list of evaluative criteria) for this assignment. Also review eCornell's policy regarding **plagiarism** (the presentation of someone else's work as your own without source credit).

Back to Table of Contents

Module Wrap-up: Developing Use Case Behavioral Diagrams (UCBD)

You now have a solid start on developing clear requirements for your system. In working on your UCBD you have determined the functions your system must be able to achieve in each use case. You also have started to formalize the relationships between operator, system, and other elements external to your system.

As you continue to work on populating your UCBD you will discover how essential this tool is for properly communicating your system's functions to others.

Back to Table of Contents

Module Introduction: Requirements That Best Serve Your Project

You have a Use Case Behavioral Diagram for your system. But are your requirements written in a formal, verifiable manner? With your UCBD in motion, you should always be looking for ways to improve and clarify the information, specifically with regard to your requirements. In order to do this you want to describe your system's use case requirements functionally instead of structurally.

In this module you will revisit the requirements written in your Use Case Behavioral Diagram to make them less structurally prescriptive. You will further explore and flesh out the responsibilities of your system, and capture these responsibilities as formal functional requirements. You will also translate the information found in a Use Case Behavioral Diagram to another format known as a SysML Activity Diagram.

Back to Table of Contents

Watch: The Importance of Clear Functional Requirements

There are key differences between functional and structural requirements. But before we look at how to write good functional requirements, it is important to understand why functional is a better option than structural.

In this video Professor Schneider describes the importance of writing clear functional requirements and how these statements shape your system. Note that in the video Professor Schneider mentions both verification and validation. It is important that you understand the distinction between these two terms. Verification involves testing your system against the design parameters. By contrast, validation has to do with how well the system meets the needs of stakeholders. Both are important, but verification is essential for proving you have met contractual obligations.

Video Transcript

There's some important guidelines, and in some cases very hard rules, to consider when writing requirements. But before we delve into them, let's look into the many uses requirements have throughout the design process. This will hopefully help us to understand why we emphasize writing them the way that we do.

Now, one of the primary uses and more obvious uses for requirements is helping to determine design specifications. Or said another way, helping to define what any valid solution must do. Now, no one person is an expert in every aspect of the needs that need to be designed. However, not everyone on a team typically has time to delve into writing good requirements. They just want the requirements to be handed to them so they can go ahead and solve a technical problem. It's something that needs to do X, Y, and Z. "Great. I can build that." So, the requirements define what the X, Y, and Z is. And it's our job as systems engineers in order to define what the X, Y, and Z specifically is.

Now, with that, a key thing is that you want to make sure that you don't prescribe exactly what the solution must be. Again, we want to write things functionally. So, you don't want to say, "the system shall have a steering wheel." That prescribes a very specific solution. Instead, to write this functionally, we want to say, "the system shall have a user interface that allows a driver to quickly and accurately control the direction of the vehicle." Then the engineer can use all their creative talents to vet something that meets that need. This is often summarized as writing something functionally, and again, not structurally, against the functionality that we really want to get at in writing our requirements.

Another key element that, a part of the design process that requirements are used for is responsibility allocation. We just figured out what needs to be designed. We now say, great, here's a set of requirements that your team is responsible for. Here's the set of requirements that your team is responsible for. You have to achieve this functionality. This is the set of functionality you're responsible for, overall. That helps the process team to go forward and keep continuity.

With that, they are also often used in the final step, which is the testing and verification step. You had these requirements in the specifications, you allocated them out. Now, we're going to actually use them for accountability tracking. We're going to use them as determining whether or not our tests are a success and ultimately, the validation of our system. Basically said another way, can you verify at the end that your system meets each and every requirement you submitted to your team, to your customer, to the government, to the government when there's a lawsuit questioning that your system delivers what is promised?

Okay. Now, we're really getting at it. These requirements are very, very serious. You can actually have your entire company riding on the fact of whether or not those requirements are written in a way that you're going to be able to support appropriately. So think about writing these requirements, particularly ones you share with external groups, as writing an official design contract. You have to be very careful with what you share with the external groups and internal teams. But regardless of which one that you're sharing it with, you write these to a level where you would feel confident that you could take that list of requirements, give it out to another group of engineers, a separate contract, and it wouldn't matter what solution they came back with, as long as it met that set of requirements that you provide with them, you will be satisfied that that was a valid solution.

And if you can write your requirements to that level, we call that you have achieved requirement completeness. Now again, it may take many iterations of the requirements in order to make that happen, but that's what we want you to be able to do.

Back to Table of Contents

Watch: Writing Good Requirements

The trick to writing good requirements is to remember that it's not so much about what your system is; it's about the functionality of your system. What needs will be met by your system? What else must your system be able to do? It's a specific way of thinking, a look at how your system will be designed to have the ability to perform a specific function. By providing good, clear requirements around the functional requirements of your system, your team now has a clear design target.

In this video Professor Schneider builds on his catapult example to show the difference between structural and functional requirements.

Video Transcript

Knowing how to write good functional requirements is one of the most essential, yet challenging engineering abilities to develop. It's like how to throw a good punch. Even the world's greatest fighters trained and refined this fundamental skill. In fact, for many people who are just beginning to write their functional requirements, they have a hard time distinguishing between what makes a good requirement and what makes a great requirement.

A key thing to keep in mind is good functional requirements are not actually about what your system does. It's about the functionality that needs to be met in order for something to be able to be done. As a starting point, let's take a simple example from our catapult toy to start to bring out some of this nuance. So let's imagine we start with requirements. We say, the system's arm fires a projectile. Sounds good at first, but this is too prescriptive. It forces us to have an arm that does the actual firing work. But what if we want to fire a projectile another way, or maybe just have the arm for show, or not have an arm at all. In all those situations we need to change this requirement in a way to allow us more freedom and flexibility.

Again, keep our focus on the functionality. So let's change it to say, the system fires a projectile. Okay, now we're moving a little bit away from the physical "how." And this is going to allow us to provide more variety of solutions. But a better, proper functional requirement might be, the system shall be able to fire the projectile. To which an experienced engineer will say, now you're getting somewhere. Well, what's the real difference? Well, this is a simple example, so not so much at first. But it's more about an approach to thinking about what's occurring.

First, let's get at the heart that the system isn't doing so much as it is being designed to have an ability. It can achieve some specific function which opens the doors to more solutions and also opens the door to more questions, such as, if the system is able to do this, what else must it be able to do? So, for example, if it was going to be able to fire the projectile, the system should be able to hold a projectile to launch, shall be able to be triggered by the user, shall be able to eject a projectile from itself. It has all these other functionalities that we start to recognize need to be achieved as well.

Then other questions begin to come out. Are there other functions that need to be occurring at the same time? And ultimately, you want to be able to feel that you have completed the list of functionality and delved into those questions deep enough that, again, you can hand off that list of requirements for anybody else to design something to meet that set of requirements and you would be satisfied with what came back with that, so long as it met those requirements. Doesn't matter how they did it, as long as they met that set of requirements that you created.

So here we really begin to notice the benefit of thinking functionally. Again, you're not defining your system as a thing, you're defining it as something that must achieve a collection of functions. And the more you discover, the more clear it will be what you have to design for, and the better you'll have to find your overall challenge, and ultimately, the better you'll be able to solve that challenge.

Back to Table of Contents

Tool: Functional Requirements

When writing functional requirements, remember it's not what your system does, it's about the functionality that's needed to be able to do it.

As you review your UCBD, make sure that you've written your requirements "functionally, not structurally." This will become natural in time. But when you are starting out, here are a few comparisons that may be helpful:

☆ **Key Points**

Focus on the **function** that your system must accomplish when writing functional requirements.

As you focus on functional system requirements you will discover more needs, which in turn will lead to more defined requirements and further clarity about how your system solves the challenge.

Thinking *Functionally* is about:		Thinking *Structurally* is about:
What is the need that has to be met	—	How you are going to meet that need
What something must be able to do	—	How you are going to do it
How should various systems interact with each other; i.e. what must each subsystem be able to do	—	The implementation that handles the interaction
How do you measure your performance	—	The actual solution's performance
Anything that can meet this description is a valid solution	—	A very specific solution

For reference, you can print this **Thinking Functionally reference sheet** as an aid while writing your functional requirements.

Back to Table of Contents

Activity: Make Your Requirements Functional

In this activity you will revisit your requirements and examine how you can improve them to further clarify the functions that your system will perform.

Task: Make Requirements FunctionalTask: Make Requirements Functional
This is Complete When...This is Complete When...
Worked ExampleWorked Example
Back to Table of Contents

Discussion: Viewing Requirements From an Outside Perspective

It can be hard to avoid getting lost in our own requirements and forgetting that our system actually needs to be implemented at some point. Often the outsider's perspective reveals something unexpected about our system, or about our approach to documenting it.

In this discussion you and your fellow students will share requirements you have developed and begin to think about your functional requirements from a structural perspective.

Instructions:

You are required to participate meaningfully in all discussions in this course.

Discussion topic:

Imagine you are at the stage to ask a contractor to create something that performs just the functions you wrote and nothing else.

1. Create a post in which you share your system's functions in the form of a UCBD. Your UCBD should include at least 3 system requirements. The UCBD you use can be either from Part One of your Course Project or another UCBD you have created.

   In addition to posting your requirements, respond to the following:

   - If the system has *to be able* to do this, what else must it do?
   - Are there other functions that are occurring at the same time?
   - Are the requirements focused on what needs have to be met, what the system must be able to do, and how the parts of the system interact?

2. After you have posted your functional requirements, select another student's post and explore what direction you would take their system based on the requirements shared.

   Reply to at least one other student's post. In your response:

   - Respond with structural solutions to their functional requirements (these can be "blue sky" structural solutions based on your best understanding of the subject material).
   - Say whether you believe you could create something that meets all the needs shared in their use case.
   - Make recommendations you think would improve the use case.

To participate in this discussion:

Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, please review eCornell's policy regarding plagiarism (the presentation of someone else's work as your own without source credit).

Please note:

While discussion board postings will be accepted through the end of the course, we strongly encourage everyone to move through this course as a group. As such, postings that are made to this board after 5pm ET on the due date will be read and graded but may not receive a response on the board from your facilitator and/or your peers. Please let your facilitator know if you have any questions.

Back to Table of Contents

Watch: Following the Requirements Rules

Writing good requirements about your system will lead to a clear vision for yourself and others. In order to provide these good requirements there are certain rules you should follow.

In this video Professor Schneider runs through these rules and discusses their importance in defining your system's requirements.

Video Transcript

To help you make sure that your requirements that you write are seen as being really professional, let's go through eleven good rules that help to serve as a good check.

Now, the first rule that we have is write shall statements. Meaning that the word shall is used in the requirement. The system shall do X. It's definitive. The word shall shall be in every requirement.

Rule number two, be correct. What you're saying is accurate. You want to make sure that this is checked against not only what your team is saying, but also potentially what your customer is saying, and that you've got the wording right in that requirement.

Rule number three, write shall statements.

Rule number four, be clear and precise. Only one idea per requirement. If you have the word "and" or a similar conjunction in the requirements, consider splitting it up into two or more requirements. This is also very important for assigning requirements and for testing purposes later on.

Rule number five, write shall statements.

Rule number six, be unambiguous. You make sure that there's only one way that you can interpret that requirement.

Rule number seven, write shall statements.

Rule number eight, be objective. Make sure that what you're writing is non-opinionated. Make sure that the words you're using are non-opinionated words, in general.

Rule number nine, write shall statements.

Rule number 10, be verifiable. Make sure that there is some measurable way that you can say that this requirement is met.

Rule number 11 is, of course, be consistent. Do not contradict another requirement. This is particularly important when you're working in very large teams where you could have as many as 10,000 different requirements, overall. So, make sure that you're always very consistent across all your requirements.

If you haven't noticed, writing your requirements as shall statements is kind of a big deal. Many agencies will not even accept requirements written any other way, despite how good the ideas are. So, again, make sure you write your requirements as shall statements. But with that being said, it's really other concepts that are truly important to make sure that your system requirements aid you throughout your design process.

Want to review the video's information? Select the button below.

Requirements GuidelinesRequirements Guidelines

Back to Table of Contents

Quiz: Bad vs Good Requirements

Now that you know how system requirements should be stated, identify what makes the following requirements *bad*.

In this quiz you will examine requirements written for a system that provides opportunities for child's play within a pirate ship theme. In each a case the requirement provided will have an issue. That is, it will be flawed in one or more ways. Your task will be to identify the flaw or flaws with the requirement.

In some cases you will be required to identify more than one flaw with the requirement as written.

This is a graded quiz. You are encouraged to take it as many times as you need in order to achieve the maximum score.

Please complete this activity in the course.

Back to Table of Contents

Watch: Setting Up Requirements Tables

Given the importance of your system's requirements in the project definition, there is value in copying requirements from all your UCBDs to a requirements table. A requirements table is a place to reference and track requirements that are appearing in other documents and diagrams. Each requirement is assigned a unique ID. This unique ID identifies the UCBD that is the source of that requirement. The ID can also signify whether this requirement is being requested internally or externally. In addition, the ID can indicate when the requirement was developed during the design process.

In this video Professor Schneider describes how to fill out your requirement table and assign unique IDs to each requirement.

Video Transcript

Requirements are typically recorded in a separate requirement table. This table often has three columns. The first column is used for a unique ID, the second column is usually written with the full shall statement requirement, and the third column is used for a short abbreviated name for that requirement, often a unique name. And again, this is a way to refer to this requirement in other different kinds of diagrams and documents.

Sometimes that unique ID also signifies whether something is called an originating requirement, perhaps a derived requirement, implied requirement, performance requirement, or any other number of designations. The exact difference between these designations can be actually different from different organizations, but they are often usually used to indicate what is the source of the requirement, whether this is a requirement that was shared with an external group or used only internally, or at what point in the design process this requirement was developed.

For our purposes, we're going to label all of our requirements as originating requirements, and hence an OR is used in all the IDs. To fill out your requirement table, simply go to the Use Case Behavioral Diagram, look at the system swimlane that you have developed already, and for each statement, or requirement, rather, that you have in that column, that should be a separate requirement in your requirement table.

Note that while there are rules for index values and for requirements, there are no widely accepted rules for abstract names. Typically, though, abstract names are kept short and unique as a best practice.

Tool: Requirements Table Template

Use the link on this page to download a template for your requirements table. You will use this tool to track your system's requirements. Each statement will contain a unique ID, the full "shall" statement requirement, and a short, abbreviated name.

Download the Tool

Requirements Table Template

Note: This is a Microsoft Excel workbook. One sheet is the template you can use as a resource to populate your own requirements table. The other sheet includes a sample of a populated table. In this sample the unique ID begins with the letter code "OR" for Originating Requirements.

Back to Table of Contents

Watch: Balancing the Spirit of the Rules with the Letter

You want strong, verifiable requirements that can be tested. But what happens when you encounter a situation where it is difficult to clearly state a testable requirement?

In this video Professor Schneider provides an example of how to balance the spirit of the rules with the letter of the rules by developing thoughtful requirement constants.

Video Transcript

Rarely is the person who writes the requirements also the person who is developing the design. So it brings up the question, how do you express both the reasoning behind why the requirement was created in the first place and still make sure that it is every bit as verifiable as you need it to be? Let's go through an example with this to both demonstrate the problem as well as a way to actually handle the situation.

So let's imagine we have the requirement, "the system shall be light enough for a 5 year old to carry." That's good because it helps to be able to express why do I care that it be light. However, the strength of a 5 year old can vary considerably. So if you're the engineer given this requirement, you should be worried about how this is going to be tested. What you really want to know is how many pounds should it be, or what is the maximum weight that can be allowed. You want a nice, strong, verifiable, testable requirement. So, you'd like something like, "the system shall weigh less than 10 pounds." That's really easy to measure.

But at the same time, if you were handed that requirement, you might wonder why, and does it make a difference if it's 9.99999 pounds, or would 5 pounds actually be much better? You might also build something that could be cumbersome while still meeting that requirement. Plus, there's another problem. Perhaps you don't know what that weight limit is. This could be especially true when you're first writing that requirement. You just know that a 5 year old has to be able to carry it.

So, to deal with this conundrum you can use requirement constants. They're similar to constants that are used in equations or in software code, basically, as a placeholder that represents a certain value. So if you don't know that 10 pounds is the limit right now, you could write something such as the system shall weigh less than X, where X is your constant value. Here, we are now able to define the value for X at some later point without actually having to rewrite the requirement.

But that still doesn't address why the system weighs less than X. So to handle this, don't use the actual term, "X." Call the requirement constant something like "can be carried by a 5 year old." Now we've added, although it's a much longer term, we have a whole requirement that is read as, "the system shall weigh less than can be carried by a 5 year old." This carries both the spirit and the verifiable letter that we need a requirement to have.

Notice that the use of underscores between the words is a throwback to constants in software code. Likewise, you'll also notice that sometimes these constants are written in camel case.

Back to Table of Contents

Activity: Use a Constant to Resolve a Requirements Issue

In this activity you will explore questions around a statement and consider ways to improve that statement for a small toy catapult.

Consider the statement:

"The system springs shall store the energy inputted by the operator."

Think through how you would answer the following questions.

Check your understanding by selecting the reveal button after each question.

1. What is the issue with this statement?

IssueIssue

2. How might we rewrite the requirement to fix the issue?

RewriteRewrite

3. How does this rewrite using a requirement constant fix this issue?

DescriptionDescription

Back to Table of Contents

Watch: Recording Your Constants' Definitions

Requirement constants will increase the efficiency with which you state some of your requirements. However, when you use constants you also need to create a requirement constants definition table.

In this video Professor Schneider describes what information you need to include in your requirement constants definition table.

Video Transcript

If you're going to use your requirement constants in the definition of your requirements, you also need to have a requirement constants definition table. This table is required to have at least three columns. The first column is the actual constant name. The second column is the value that the constant represents. And the third column are the units that go along with that value.

It's not uncommon that you may see additional columns added as well, one of the most common being an additional column that says whether or not this value is an estimation or whether it's a final value. Similarly, you may find additional columns that say, is this estimation going to be updated by a certain date and what is the final date that this estimation will be determined. You may see some additional columns as well that indicate the source of the value that was chosen, whether it be from a customer, whether it be from a government requirement, etc. You may also notice that a name of a person who is responsible for setting that. So you can ask that person, why was this value chosen? And you may also notice an additional column just for notes that people may want to include as part of that as well.

Now, like constants in code, some constants can be used in multiple requirements. Hence, in these cases, requirement constant tables often require columns for every requirement designated by their requirement and unique ID. Then you mark an X in the column for every requirement that that constant is used in. This way, should the value of that constant ever be changed, you know what requirements will be influenced, instead of having to search through all the requirements manually.

In some cases, a requirements definition may actually require more than one value. Let me give an example. The system shall meet reliability standards. Reliability standards being the constant. In this case, the constant definition table will have a value listed in the column that

is actually a location where that document can be found that defines those standards. Now, there are different mindsets of whether or not this is acceptable, but it's not uncommon.

Back to Table of Contents

Tool: Requirement Constants Definition Table Template

Use the link on this page to download a template for your requirement constants definition table. You will use this tool to track the constants used in your requirements.

Download the Tool

Requirement Constants Definition Table Template

A constants definition table should include a column for the constant, another for its value, and a third for its unit. Other columns might indicate such data as: estimated progress, date of most recent update, date finalized, source, ownership, or notes.

Note: This is a Microsoft Excel workbook. One sheet includes the template you can use as a resource to populate your own requirement constants definition table. Another sheet provides a description of the possible columns to be used in your table. Two additional sheets provide samples of populated tables. In particular, the Sample 2 sheet shows how to track which requirements rely on which constants.

Back to Table of Contents

Watch: Delving Into Your Requirements

Now that you understand all that goes into writing good requirements, it's important to go through your UCBD again and look for ways to further develop and expand your original requirements. Often there are hidden functionalities, and as you start to look deeper at what your system can do, more requirements emerge.

In this video Professor Schneider provides an example in he which looks closer into the functional needs of a system and demonstrates how the process of delving provides significant insight.

Video Transcript

The key to good writing is often good rewriting, and this is particularly important when developing requirements. So right now I'd like to go through with you an example on how to iterate, and really delve into your requirements to help discover all the functionality that might be hidden within because normally when we start to write requirements we tend to write them at a rather high level. And it may sound good at first, but we have to recognize that in order to do that high level requirement, there's many other kinds of functionality that need to be identified and formalized as well.

In the example I'm about to show you, not only are we going to discover additional requirements and additional functionality, we're even going to identify potentially three other major use case scenarios that we're going to have to handle, starting with just one single requirement. So, let's imagine I've got a robotic arm of some kind that I'm designing and this arm is being designed to actually pick up an object off a desk. So, I might have a requirement such as, the arm shall be able to pick up the target object on a desk. Sounds like a good requirement to begin with. Nothing wrong with it at face value.

But as I start to think about it, if I'm going to have to meet this requirement, again, what are the other functionalities that must be achieved? I recognize that in order for my arm to be able to pick up that object, I also have to achieve the functionality of identifying that object, potentially apart from other objects. I also have to achieve the functionality of being able to determine the location of that target object in 3D space.

This process continues to explore, what are other ways or other functionalities that I need to achieve in order to achieve that first task? Such as, the arm end effector has to be able to conform to that target object well enough in order to actually lift up that target object. That's

another kind of functionality. Another functionality that must be achieved is that I shall not damage that target object. The idea of damage. That's an interesting idea. How am I going to quantify that? Sounds like we could probably use some requirement constants to begin with at this point, but at some point, I'm going to have to define damage very formally, overall.

As I explore further I recognize that another functionality that comes out of this is I have to determine the actual angles of all the joints in my arm in order to pick up that target object. And I have to have some kind of means to be able to calculate a path of motion, overall. So, there's some additional kinds of functionality. And as I think about that, I recognize well, okay, I'm going to have to use inverse kinematic equations in order to make this happen. I'm going to need models of this arm at some point. And if I think about how I'm going to develop those things, I recognize, you know what, I probably need some additional information here about my design, overall. Do I have any constraints on how smooth this motion has to be? What is the desired rate of motion that I have for this, overall? So, I'm going to have to go back and ask a lot of questions, potentially even of my customer in this. Again, just in analyzing the kinds of functionality and what is necessary in order to achieve those kinds of functionality.

We can go even further to think about, alright, not only do I need to be able to pick up that target object, but I need to be able to avoid any obstacles as well as I reach for that target object. And if I'm going to avoid those obstacles, well, I can delve down even deeper for more kinds of functionality to say, I've got to be able to identify those obstacles, and I have to determine those obstacles' location in 3D space.

So ultimately, even though I started out with what sounds like a pretty good requirement, I did identify three major additional kinds of functionality that need to be achieved. I've determined that there's lots of things that have to deal with the detection of different objects, both the target object and obstacles. I've determined that there is additional functionality with actually grabbing the object, and I've determined that there's a lot of functionality that has to do with arm motion. All of which can be thought about as different kinds of series or use cases that I'm going to have to go into more detail with.

Again, I started out with just one requirement that sounded pretty good, but in order for me to really make my design work, I'm going to have to address all these other kinds of functionality. And that's why it's so important to always delve as much as you can into your requirements.

Back to Table of Contents

Read: Tips For Delving Into Requirements

Our example of the robotic arm shows how this act of delving into the functional needs can lead to more significant insights. From the requirement, "The system shall be able to pick up the target object on the desk," a large number and variety of additional requirements were discovered.

☆ Key Points

Delving should be done after the initial round of requirements writing.

Delving into your requirements helps you discover additional requirements to include.

As you add requirements to your UCBD you may discover that different kinds of functionality support one initial requirement. This discovery may evoke the original UCBD to be split into several UCBDs.

The system shall be able to...

Identify the target object apart from other objects*	Determine location of the target object in 3D space*	Determine the proper arm angles in order to pick up the target object***
Detect obstacles*	Conform to the target object shape well enough to lift the target object**	Calculate a path of motion***
Determine the location of obstacles in 3D space*	Not damage the target object**	Avoid obstacles in reaching for the target object ***

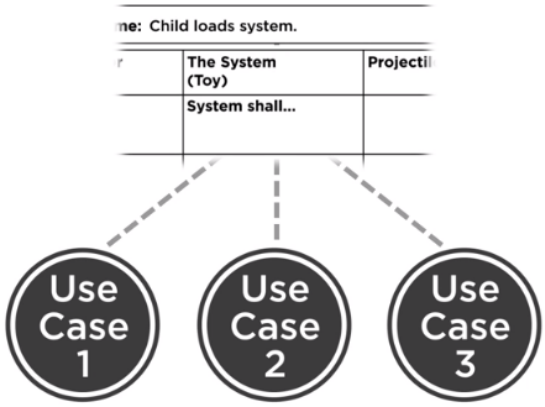
Through this act of delving, there were a lot of different kinds of functionality going on to achieve that one initial requirement.

This can inform you that you may want to split your original use case into several; for example, we may take this list of new requirements and label each with a single*, double** or triple*** asterisk.

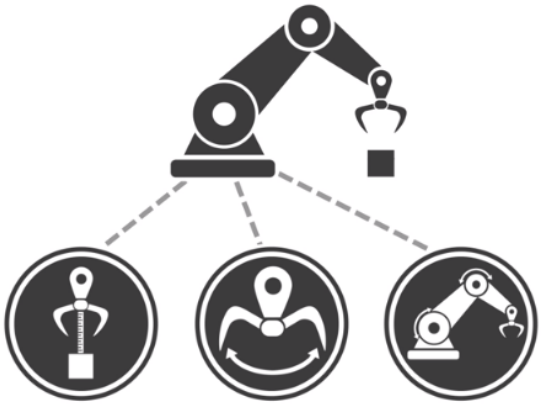
* "System identifies objects"

** "System grabs target object"

*** "System changes position"



Grouping in this way allows you to explore each use case further to see if there are any additional functional requirements to consider in your design. Furthermore, it also helps you begin to recognize more precisely what you need to do. In this example you may now recognize this might involve a significant sensor sub-problem (and maybe even a computer vision sub-problem). You'll similarly expect to have to do significant mechanical modeling of your system and probably work with inverse kinematic equations and controller designs. You may also realize this is a lot of work and it will inspire you to come up with a different way to meet your customer's needs. Either way, you are beginning to get a better sense of the possible scope of work and skills needed to be a success.



Also notice that even though some of the requirements marked with a ***** were discovered by delving into the later ******* requirements, we decided to group these ***** requirements together. This informs you as a designer that instead of just focusing on identifying the target object, you now might consider trying to create something more generic to distinguish the target from other non-target objects. Had you not explored your functional requirements first, you may have begun working on a more optimized solution to find just the target object. And then when you learned later on that you also had to handle detecting other objects, you may have had a lot of rework to do: "Man, if I had only known that was part of the problem from the start, I would have done things completely differently."

This is just one example, but hopefully it begins to hint at the value of defining your system functionally first.

Back to Table of Contents

Activity: Identify Missed Functionality

In this activity you will continue building your Use Case Behavioral Diagram by identifying missed functionality that still needs to be captured.

Task: Identify Missed FunctionalityTask: Identify Missed Functionality
This is Complete When...This is Complete When...
Worked ExampleWorked Example
Back to Table of Contents

Activity: Rewrite Problematic Requirements

In this activity you will explore questions around requirement statements for a small toy catapult and consider ways to improve each requirement. For each of the examples think carefully about your answer before revealing the prepared response. The practice you get analyzing the requirements is just as important as the answer you come up with.

- **Example 1**
- **Example 2**
- **Example 3**

Consider this statement:

Example 1

"The system shall lock the receptacle in place."

Think through how you would answer the following questions.

Check your understanding by selecting the reveal button after each question.

1. What is the issue with this statement?

IssueIssue

2. How might we rewrite the requirement to fix the issue?

RewriteRewrite

3. How does this rewrite address the issue?

DescriptionDescription
Consider this statement:

Example 2

"The system shall detect the command from the child to release."

Think through how you would answer the following questions.

Check your understanding by selecting the reveal button after each question.

1. What is the issue with this statement?

IssueIssue

2. How might we rewrite the requirement to fix the issue?

RewriteRewrite

3. How does this rewrite address the issue?

DescriptionDescription

Consider this statement:

Example 3

"The system shall eject the contents of the receptacle."

Think through how you would answer the following questions.

Check your understanding by selecting the reveal button after each question.

1. What is the issue with this statement?

IssueIssue

2. How might we rewrite the requirement to fix the issue?

RewriteRewrite

3. How does this rewrite address the issue?

DescriptionDescription

Back to Table of Contents

Discussion: Fixing Requirements

You've had some practice identifying problems with requirements and rewriting the requirements to address the issue. In this discussion you'll work with the other students in the class to find the best refinements for each of several requirements.

Instructions:

You are required to participate meaningfully in all discussions in this course.

Discussion topic:

To begin, select one of these original statements.

- "The toy's cannon will be able to shoot short and long distances."
- "The toy shall be able to be completely submersed in up to 5' of water and not sustain damage."
- "The toy shall be fun to play with."
- "The toy cannon shall shoot no further than two feet, and the toy cannon shall be able to shoot at the long length of three feet to within four inches."

Next, identify an issue with the original requirement statement you've chosen.

1. **Create a post** in which you:
   - post the original statement
   - identify an issue with the original statement
   - provide a rewrite that addresses that issue
2. Now select **someone else's rewrite** and describe how you believe their rewrite addressed the original issue. Or, if relevant, share how an additional rewrite could further improve the requirement.

To participate in this discussion:

Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, please review eCornell's policy regarding plagiarism (the presentation of someone else's work as your own without source credit).

Please note:

While discussion board postings will be accepted through the end of the course, we strongly encourage everyone to move through this course as a group. As such, postings that are made to this board after 5pm ET on the due date will be read and graded but may not receive a response on the board from your facilitator and/or your peers. Please let your facilitator know if you have any questions.

Back to Table of Contents

Read: Creating UCBDs for Your Other Use Cases

Now that you have completed a diagram for one of your high priority use cases you are ready to repeat these steps for your remaining high priority use cases and some of your medium or even low priority uses cases. How long you spend on developing use cases depends on the scope of your project and the time you have available. You most likely do not have the time to create UCBDs for all of your use cases. So when choosing which lower priority use cases to explore, select use cases you think will involve different kinds of functionality and will uncover system requirements that are not yet declared.

It's perfectly fine to set additional requirements for your system, even if they aren't directly from one of your UCBDs. If it's an important function that would probably have shown up in one of the UCBDs you didn't have time to create, you can always add it to the requirements table anyway. In these cases though, be sure to record where the inspiration for the requirement came from. This is done in case, for example, the source of the inspiration changes. You will then know to either alter or eliminate that requirement from your requirements table.

Note: While you will need to create a number of use cases in actual system development, for the purposes of this course you will not be asked to complete any additional use cases. The amount of time required to develop each additional use case is often roughly equivalent to the time spent developing your first use case, because iteration is essential.

Back to Table of Contents

Watch: Populating Activity Diagrams

You may be familiar with the Systems Modeling Language (SysML, for short) due to a workplace implementation or from a documentation requirement in one of your projects. SysML has long been popular with government and large-scale project developers, and it is increasingly popular even with smaller organizations. To ensure your work is seen as professional, you should become familiar with SysML documentation. And even if your own work doesn't utilize SysML you may run across a SysML document at some point.

The analogue of a UCBD in SysML is to use an Activity Diagram. The information you capture in a UCBD is represented slightly differently in an Activity Diagram, but many of the important features are the same. In order to help you become familiar with the format you will be creating an Activity Diagram based on a UCBD in this course.

In this video Professor Schneider describes the components of an Activity Diagram and relates it the the UCBD you are already familiar with.

Video Transcript

SysML is a common set of standards used by many systems engineers around the world. Now, when using SysML, they represent some of the diagrams that we've gone over in different ways. For example, a Use Case Behavioral Diagram is represented as an activity diagram in SysML. The ideas are very, very similar between the two different forms. In fact, most of the changes are largely superficial. But nevertheless, I'd like to go over the differences with you, so that way, either you can create one in SysML if you desire, or at least be able to interpret somebody else's diagram if presented with one.

Now, in SysML, the way that they represent the different swimlanes that we went over before in a Use Case Behavioral Diagram is as activity partitions. They look very much like the different columns that we used before, but again, the official term is activity partitions. They also are sandwiched in the same way that we would put beginning and ending conditions on our Use Case Behavioral Diagram, except they're referred to as precondition and postcondition constraints, containing all the same information, but just represented, again, in a slightly different way.

One other thing that you'll notice is that instead of including different statements on different rows, they include what are referred to as actions, which are basically small boxes that are, again, all kept just one per, basically, line of your diagram. There's not well-distinguished rows,

per se, but per line, still one action per line. And then these actions are connected with what they refer to as control flow arrows, in order to indicate how you move through the diagram, overall. Basically, again, it reads the same as if it was on a spreadsheet, but the appearance is slightly different.

They also include terminal nodes at the beginning and end of the overall body of the Use Case Behavioral Diagram, or again, in this case, activity diagram. And there is space at the bottom that can be used for additional notes as well. However, the one slight difference that is important to recognize is that in the actions that you are writing, actually formally referred to as opaque actions, the terminology is written slightly differently. Unlike in the Use Case Behavioral Diagram we showed, in the swimlane that you use for your system, or the activity partition using your activity diagram, the requirements are written in SysML as informal statements. I know that may seem like sacrilege at this point. But when writing an activity diagram, you use informal statements in all parts.

So, how do you reach the formality that's needed so badly for the requirements? Well, in SysML, for each one of those opaque actions that you have for your system, you also have to include that requirement in a separate requirement table. And a requirement table in SysML is, again, another official document that you can officially link each one of those requirements table entries to those opaque actions in your Use Case Behavioral Diagram.

In the video you see a Requirements Table that translates the informal statements in an Activity Diagram to the more formal requirements you're used to seeing in a UCBD. Note that in some software packages what is shown in the video is referred to as a Requirements Diagram. In these software packages a Requirements Table refers to a spreadsheet form of the same information.

Back to Table of Contents

Tool: SysML Activity Diagram Template

Download the Tool

SysML Activity Diagram Template

SysML Activity Diagrams are a standard communication tool among many top teams. Like a UCBD, a SysML Activity Diagram shows the flow of a use case scenario. This diagram provides an explicit visual explanation of how the use case moves through the requirements. For more specifics on those requirements, an accompanying SysML Requirements Table provides the system requirement as formal statements.

This is a template for your SysML Activity Diagram. You will use this tool to help you create a SysML variation of a Use Case Behavioral Diagram.

Note: This is a Powerpoint template for your SysML Activity Diagram. The accompanying SysML Requirements Table is provided in one of the slides. Samples are also provided as separate Powerpoint slides. You can use the Powerpoint template provided or create the Activity Diagram and/or Requirements Diagram in an alternative program.

Back to Table of Contents

Assignment: Course Project, Part Two—Improve Your Requirements and Create an Activity Diagram

For this portion of the project you will revisit the Use Case Behavioral Diagram from Part One of the project and delve into your requirements.

You will examine which of your statements are structural versus which are functional, and then you will rewrite your structural statements so that they are formal, verifiable statements. With your completed UCBD you will create a SysML variation of an Activity Diagram and a corresponding Requirements Table for that use case.

Use the **SysML Activity Diagram Template** to provide an Activity Diagram and Requirements Table for your use case. Save the file with your last name appended to the filename.

You will need to turn in your **updated UCBD** along with your **SysML Activity Diagram** and **Requirements Table** into your instructor as part as your project completion.

Completion of all parts of this project is a course requirement.

Instructions:

1. Open your saved course project document (if needed, **download the course project** again now).
2. Complete Part Two.
3. Save your work.
4. Submit your completed project for grading and credit.

This is a required part of the final course project; completion and submission of all parts of the course project will be required in order to achieve credit.

Before you begin:

Before starting your work, please review the **rubric** (a list of evaluative criteria) for this assignment. Also review eCornell's policy regarding **plagiarism** (the presentation of someone else's work as your own without source credit).

Back to Table of Contents

Module Wrap-up: Requirements That Best Serve Your Project

In this module you explored and fleshed out the responsibilities of your system. You examined your system requirements and rewrote them as necessary to remove structural specificity. With these responsibilities captured as formal functional requirements, you delved further into your requirements. This delving allowed you to discover additional functionality needs. With a more complete set of requirements in place you used one of your system's use cases to create a SYSML Activity Diagram.

You now have a Use Case Behavioral Diagram (UCBD) for one of your system's use cases, and you have a SysML Activity Diagram that represents this UCBD. Most importantly, the requirements in these documents represent the functions your system must be able to achieve throughout the use case.

Back to Table of Contents

Thank You and Farewell

David R. Schneider
Senior Lecturer
College of Engineering
Cornell University

Congratulations on completing *Developing System Requirements*. As you have seen, determining the functionality of your system's requirements is essential throughout each use case. Creating strong formal requirements will take you and your team a giant step forward in designing your effective system.

From all of us at Cornell University and eCornell, thank you for participating in this course.

Sincerely,

Professor David R. Schneider

Back to Table of Contents

CESYS522 Glossary

Click here to view the course glossary:
https://s3.amazonaws.com/ecornell/content/CESYS/CESYS522/CESYS522_glossary.pdf

Back to Table of Contents

Tool: Course Tools

Defining Your System Requirements Checklist

Defining Your System Requirements Guide

Use Case Behavioral Diagram Template

Thinking Functionally Reference Sheet

Requirements Table Template

Requirement Constants Definition Table Template

SysML Activity Diagram Template