Developing Your System Requirements
Use Case Behavioral Diagrams

Knowing your stakeholder's perspectives, the context in which your system must operate within, and the use case scenarios that your System must accomplish begins to establish what parameters you will have to design to. But what truly defines your system is what functions it must perform in order to successfully accomplish these use cases. If you're able to formalize these functional requirements, you have created a technical definition of what any valid solution to your problem must do. Thereby you have also taken a critical step towards enabling every designer to both understand what your system must accomplish while still allowing them the opportunity to still showcase their talents at developing a creative and valued solution towards meeting these requirements.

This guide helps you analyze the use cases and formally determine functional requirements that are required of any valid solution to the challenge you are trying to solve. More so, this guide also shows how this process can indicate functions you will want to measure the performance of and introduce techniques for brainstorming how you can measure that performance objectively. Once you have taken these steps you will have a far greater understanding of what it is you must achieve and you will be able to create a far more successful, complete, and efficient solution than you would have been able to otherwise.

Before you begin: (you can think of this as Step 0) If you haven't done so already, consider who are your stakeholders. Stakeholder is a catch-all term used to define those who have any influence on what or how you design something but is better to be thought of as anyone or anything your design affects. You can think of those affected as primary stakeholders or secondary stakeholders.

For the primary stakeholders, think about what they need your system to do (or not to do). If you are working with a specific client, you also want to define your client's deliverables. For secondary stakeholders, who may not be directly associated with using your system but are strongly influenced by it or its outcomes, ensure you aren't accidentally violating any of these stakeholders' needs.

Step 0 is complete when you have defined your stakeholders and your system's context and interfaces, have prepared a list of use cases to explore, and (if applicable) have a defined list of deliverables for a specific client(s).

Step 1: Rate your use cases to the best of your current knowledge as to which ones are the most important. A typical rating system is simply to rate them as high, medium, and low priority. Higher priority ones are use cases that may:

- directly address stakeholders' primary needs,
- have a significant influence on the performance of your system,
- address situations that are considered high risk, including use cases that focus undesired occurrences,
- anticipate your System will be required to perform complex and important tasks, and/or
- occur very frequently.

There may be other reasons as well, and likely you have not formally defined what constitutes "high risk," or established all the ways your System's performance — and thereby value — could be evaluated, but at this point that's okay. From what you do know or assume at this point, do your best to identify higher priority use cases. You can always decide to change these priorities later; this just gives you a starting point.

Step 1 is complete when you have prioritized your use cases in some way that identifies which use cases you currently consider to be the higher priority.

Step 2: Select one of your higher priority use cases to further explore. Start to create a use case behavioral diagram (e.g., use case matrix, use case swimlane diagram, use case activity diagram). A use case behavioral diagram — abbreviated as UCBD in this guide — is a way to describe step by step what your System must *functionally* do and what other objects, users, etc., are doing during that use case.

The file, **UseCaseBehavioralDiagramExample.pptx**, is provided to help demonstrate how one might complete a UCBD for a toy catapult-like System called "Child uses toy catapult." This use case is high-level and generic, but for a simple toy catapult it's sufficient for demonstrating the UCBD process.

For now, copy the template in the first tab of the sample and change the name label in the upper left hand corner to your use case name.

Step 2 is complete when you have copied the template or created a similar UCBD setup and have labeled it with your use case name.

Step 3: To begin creating your USBD, determine who or what are the main *actors* involved in this use case. Actors is a generic term in USBDs used to typically mean anything that interacts with your System. Many of the items in your Context Diagram can be considered actors.

Your *main actor(s)* are commonly human operators or users but could be another machine that is sending commands to your System. A main actor is also sometimes thought of as the source of the triggering or perhaps guiding input to the use case. In the example, the Child is the main actor whose actions and needs drive the System to have to be able to do something.

Label a separate column for each actor and one for your System. Traditionally, the main actor(s) are placed in columns to the left of your System's columns. Other actors'

columns are thereby placed to the right of your System's column. These columns are sometimes also called *swimlanes* as also mentioned in Step 6.

Step 3 is complete when you have created a separate column for each actor and one for your System according to the guidelines given above.

Following standard guidelines like this make it far easier for someone else — or even yourself, years later — to be able to pick up your UCBD and quickly gain information from its formatting, such as who is considered the main actor(s).

Step 3a: Jot down "notes" as you go through the UCBD process to help provide the reader insights on any possible misconceptions. For example, one might think that the projectile is considered part of your System since many similar toys come with their own ammunition.

However, in our example, the catapult-like System should be designed to work with a variety of other child-safe toys as projectiles that are not part of your System design scope. Hence, the projectile is also considered an actor with its own column and not part of your System. Think of the situation where you would be designing a new Nerf blaster, but you wouldn't get to design the Nerf dart. That's already standardized, and since it's outside of your design scope, it would be listed as a separate column.

We also assume that the projectile is something relatively safe and not, for example, a pet gerbil. Hence, we limit our UCBD to discuss only those cases where the projectile is a child-safe toy.

To inform the reader of assumptions like these, the bottom of the UCBD is reserved to list notes for the reader. Notes are often written informally, but each one is numbered so they can be easily referenced later.

Step 3a is complete when you have added a notes section to your UCBD and any notes you think the reader may need. You may continue onto other steps and/or return to this step as needed throughout your UCBD process.

Step 3b: Create just one — and only one — column for your System. Do not split your System into subsystems at this stage. This is to help you stay with the concept that your System is not yet defined, and you want to stay in that professional designer mindset that your System is not yet named and could be anything, so long as it meets all the identified needs and functions well. And since we're in the process of identifying these functions, it's best to keep your System as just "System."

Yes, in the example, we called our system a toy-catapult, but that was done partially so it is easier for the reader to jump in at this stage of the design and follow the USBD process. It's also professionally okay at this stage of design to use a placeholder idea for your System, recognizing that you may have to throw away that placeholder idea later in your design.

Step 3b is complete when you have one — and only one — column for your System in your UCBD.

If necessary, you have accepted a placeholder concept to start your UCBD process, fully accepting that you may need to redo significant parts of your UCBDs if you later must go in a different direction and the placeholder concept was discovered to make your UCBDs too specific to that placeholder concept.

Step 4: Establish what are the starting conditions for your use case. In the example, the starting conditions are simple: "System is in the unloaded state," meaning simply that the catapult hasn't been loaded yet. There are no significant conditions that need to be stated for the other actors for the use case to begin. Yes, you could state that the child and projectile need to be near your System, but you must decide whether that information needs to be specifically stated.

You may have a starting condition, or even multiple starting conditions, for each actor and your System. Sometimes a starting condition is described as being that another use case has been successfully (or unsuccessfully) completed.

Step 4 is complete when you have established a starting condition(s) for your use case and listed it in your UCBD.

Step 5: Establish what are the ending conditions for your use case. In the example, the ending conditions are the same as the starting conditions: "The System is in the unloaded state." This doesn't have to be the case but think of the ending conditions as being how your use case scenario should end. It's typically in a stable state or it could be in transition as perhaps the stating conditions for another use case.

Step 5 is complete when you have established the ending condition(s) of your use case.

Step 6: Explain what must functionally occur from your use case's starting condition to its ending condition. Often, but not always, the use case begins with some kind of a trigger action that will cause your System to do something. Often this trigger is initiated by the main actor as is done in the example.

Actions done by any of the actors are typically written in a slightly less formal way, simply stating what is occurring. What your System should do, however, should be written formally as *functional requirements*. Below is an in-depth description of how to write good *functional requirements* and the later steps help provide explanations as to how to delve into functional requirements to truly define your System.

For now, notice in the example the following rules for this part of the UCBD process:

- Actor actions and formal requirements, referred to generically as statements, should be written in a chronological order from the starting to the ending condition.

- Each statement must be written in the column corresponding to the actor or your System who is primarily responsible for that statement, i.e., the actor's or your System's *swimlanes*.
- Each statement should be written in its own separate row; no row should have more than one statement in that row.
- The area where the statements are written in the UCBD is sometimes referred to as the UCBD *body*.

While writing formal requirements is easy, writing *good* requirements that can be of the most help is something that requires more thought and practice. Further, as many start writing requirements it's even difficult for them to tell the difference between a good requirement and a great one. Read through the example **UseCaseBehavioral DiagramExample.xlsx** and then the section below on *Functional Requirements* to help you create more meaningful and valuable requirements. Then, create your own first pass at filling out the UCBD body.

Step 6 is complete when you have a series of actor statements and System requirements that describe what must occur and the functionality your System must provide/perform in order to advance from the use case's starting condition to the use case's ending condition.

Tips for Requirement Writing

Functional Requirements: When writing functional requirements, it's not about what your System does, it's about the functionality that's needed to be able to do it.

At face value, the difference seems minute. As simple example, in describing the toy catapult use case, one might say "The System fires the projectile," but the way to write this idea as a proper functional requirement is to say, "The System shall be able to fire the projectile." What's the real difference? Not much at first, but then we ask, "If the System has to be able to do *this*, what else must it do?" the benefit of thinking functionally begins to become apparent.

If it's going to be able to fire the projectile, the System…

- shall be able to hold the projectile until launch.
- shall be able to be triggered by the user.
- shall be able to eject the projectile from itself, etc.

As you continue to brainstorm, you begin to realize all that your System must accomplish, some of which may not have been as evident at first. Before it was all just going to do it somehow, but now you know you must design something that works together to perform all these functions, and all the functions of your other use cases that you'll discover. The more you discover, the clearer it will be what you must design for. And if you know more clearly what is all that you must design for, the better you have defined your overall challenge better, and the better you will be able to solve that challenge.

More information on the formal rules for writing requirements is given in Step 10. You may want to review that step now, and then read it again later as you review your work at the end.

Delving Into Functional Requirements: Taking another example to show how this act of delving into the functional needs can lead to more significant insights, imagine that this time you were designing some kind of robotic arm. One requirement you could see being part of a UCBD might be: “The System shall be able to pick up the target object on the desk.”

As you delve into what is necessary to perform this requirement, you discover a large number and variety of additional requirements including:

- The System shall be able to identify the target object apart from other objects*
- The System shall be able to determine location of the target object in 3D space*
- The System end effector shall be able to conform to target object shape well enough to lift the target object**
- The System end effector shall not damage the target object**
- The System shall determine the proper arm angles in order to pick up the target object***
- The System shall be able to calculate a path of motion***
- The System shall be able to avoid obstacles in reaching for the target object***

And these could be delved into even further, for example, the last requirement could inspire more requirements itself such as:

- The System shall be able to detect obstacles*
- The System shall be able to determine the location of obstacles in 3D space*

And the list could go on. But even taking it this far reveals that there are 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, the requirements labeled with a single, double and triple asterixes may want to be made part of their own new use cases which could be:

* “System identifies objects”
** “System grabs target object”
*** “System changes position”

This will allow you to explore each use case further to see if there are any additional functional requirements to consider in your design. Further, it also helps you begin to recognize more precisely what you need to do. You recognize now this might involve a significant sensor if not 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 needs.

Either way, you are beginning to get a better sense overall of the possible scope of work and skills to be a success.

Also notice that even though some of the requirements marked with a single asterisk were discovered by delving into the later three-asterisk requirements, we decided to group these asterisk requirements together. As a designer, this informs you that instead of just focusing on identifying the target object, you now might consider trying to create something more generic to identify other 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 that you also had to handle detecting other objects, you may have had a lot of re-work 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 begins to hint at the value of developing your System functionally first.

Discovering Performance Criteria Through Functional Requirements: As part of discovering the functionality your System must achieve, you may also realize that it’s not just a matter of “what” it has to do, but it can make a significant impact on “how well” your system does it.

For example, take the final functional requirement in the toy catapult example: “The system shall eject the contents of the receptacle.” If presented with several toy catapults, you might judge them on how fast the projectile flies, and how far the projectile flies. Hence launch velocity and launch distance could be considered *two performance criteria*.

Measuring these two criteria might be straight forward but coming up with objective ways to measure identified criteria is sometimes not as simple. For instance, in the robotic arm example above, you could imagine performance criteria such as:

- The rate at which the arm moves
- The time it takes to move the target object from initial command to completion
- The time it takes to recognize objects
- The accuracy at which it identifies the target object
- How smoothly the arm moves, etc.

Recognizing the performance criteria is very important for informing how you develop your design. But here you’ll also have to objectively define the way you measure “accuracy” and “smoothly.” Accuracy might be measured by how often the target object is found correctly, missed, or reported incorrectly and may involve establishing a series of test scenarios that represent those various situations you may encounter.

“Smoothly” might be objectively measured by a combination of variation from an expected motion path, and/or the maximum, average, and standard deviation of acceleration measured from the end effector, and/or dissipation of residual vibration

after a motion has completed, etc. The way you objectively measure your performance criteria is call a performance metric. The “Decision Matrix Guide and Performance Criteria Tips for Deliverables” and “Decision Matrices Guide” provide more detailed information on performance criteria and how to use them well in your design process.

As part of establishing your measures and exploring your System functionality, it’s also good to ask why you care about a particular function. As another example, imagine you were designing a car-like system. You could then envision that you may come up with the requirement: “The System shall minimize fuel consumption.” But why is this functionality important? Is it to:

- minimize the cost of operation,
- minimize the environmental impact,
- minimize the fuel storage space,
- minimize the weight of the System,
- maximize the travel distance, or
- maintain a competitive level with competitors?

The answer may be all the above, but probably not all of them are equally important objectives. You’ll develop an even better design if you can not only figure out the functionality you need to achieve, but why that functionality is important, and how are you going measure all those aspects together to estimate your overall design’s performance.

Ask yourself, how do you measure your performance in any of the use cases, and how will that influence your design? Even more generally, if there’s a functionality or idea that you like, why do you like it? You may even return to discovering you need additional functions; in toy catapult example, you might recognize that if the launch velocity increases too greatly, safety is a concern. Safety is functionality and a performance criterion that you need to consider.

Step 7: Review your use cases for missed functionality and performance criteria. It’s rare that even the most experienced professional designer will think of all of the key functions or delve far enough on your first try. So it’s quite common that your first pass at a UCBD might be pretty short and rather high level. Go over it again, and maybe again after that, and keep asking questions such as:

- “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?”
- “If I asked a contractor to create something that only performed the functions I wrote and nothing else, would I be confident that what I got back would be able to meet all of needs associated with this use case?”

These are tough questions, but continue with them until you feel you can functionally define what any System must be to meet the needs of this use case. When you have reached that point — and perhaps done this for several use cases — you may find that you begin question many of your previous assumptions, even, “Does it really need to be

a catapult? Could there be other ways to meet the requirements above?" Taken together, anything that meets the collection of requirements across your use cases is a valid solution. It is only through creating more functional requirements, that what the System should be begins to take shape.

Step 7 is complete when you have reviewed your use cases and requirements using the questions provided here and the delving tips suggested previously, refining or adding any additional requirements to your UCBD.

Step 8: Review your use case one last time for any additional performance criteria and ideas on how to measure them.

Step 8 is complete when you have identified performance criteria that could evaluate the value of any solution's ability to handle this use case.

Step 9: Repeat Steps 2 through 8 for your remaining high priority use cases and some of your medium or even low priority uses cases. Most likely, you do not have the time to create UCBD for all your use cases. So when trying to choose which lower priority use cases to explore, select the ones that you think will involve different kinds of functionality and set different kinds of requirements for your System than those you have already declared.

It's perfectly fine to set additional requirements for your System, even if they aren't directly from one of your UCBD but you know it's still an important function that would probably have shown up in one of your UCBDs if you had time to create UCBD for all of your uses cases. In these cases, it is at least good to record where the inspiration for the requirement came from. This is done in case, for example, the source inspiration changes; you will then know to either alter or eliminate that requirement from your list of requirements overall.

Step 9 is complete when you have a list of functional requirements that define the functionality that your System must achieve.

Step 10: As writing requirements is considered a cornerstone of professional design by many, use the following traditional 12 properties and guidelines for writing requirements as a check.

Requirements should be:

1. Written as Shall Statements.
2. Correct: What you're saying is accurate.
3. Written as Shall Statements.
4. Clear and Precise: Only one idea or requirement. If you have the word "and" or similar conjunction in your requirements, it's considered better to split the requirement into two.
5. Written as Shall Statements.
6. Unambiguous: Only one way to interpret.
7. Written as Shall Statements.

8. Objective: Non-opinionated.
9. Written as Shall Statements.
10. Verifiable: There is some measurable way you could say this requirement is met.
11. Written as Shall Statements.
12. Consistent: Does not contradict another requirement.

If you haven't noticed, writing your functional requirements as shall statements (meaning that the word "shall" is used in the requirement, e.g., "The System shall do X"), is kind of a big deal to a lot of professional designers — in fact, many agencies will not accept requirements written any other way, despite how good the ideas behind them are. But it's the other concepts that are truly important for developing your System's requirements.

Step 10 is complete when you are confident that your requirements are written in a professional manner.

Step 10a: In reviewing your requirements, you may have noticed that it would be hard to objectively verify whether a requirement is met or not in your end system. From our catapult example, the requirement as written "The system shall be able to store the energy input from the operator" actually states that the system will accept and store **all** energy possibly input by the operator (even if they jumped on it). To make this more realistic, easier for an engineer to later design to, and to be testably verifiable, we can add a maximum threshold of energy to be stored. But what if we don't know what a good threshold should be yet?

This is a common situation that arises, and to handle this we can use requirement constants. Like constants you might write in an equation or a computer program, they are terms used to represent a value. In our example, we might re-write the requirement as "The system shall be able to store the energy input from the operator up to MaximumEnergyInput" where MaximumEnergyInput is the constant name.

Notice that we choose a constant name that captures the spirit of what the constant represents. It's okay to have long names rather than something generic like "x1" because once you have hundreds, maybe thousands of requirements, it's hard to remember what x1, x2, x3, etc. really mean. It is also common to write the constant as all one word, sometimes using capitalization to distinguish words within the overall constant name (like the idea of camelCase).

To keep track of all your constants and to define their values, you should also create a Requirement Constant Table. There are many forms of Requirement Constant Tables but as shown in the example, a good one should at least include the:

- Name of the constant
- Value of the constant
- Units of the constant value
- Source of information for why this value was chosen

As it can be common that requirement constants may be updated throughout the design process additional information could include:

- Whether the current value is an estimate or not
- The date when the value was last updated
- Which team member made the update
- When is the estimated expected to be updated next
- When is the final value due

Step 10a is complete when you have used requirement constants to improve upon your requirements, especially along the lines of the rules stated in Step 10. The addition of a requirement constant in our example has not only made the requirement more realistic, but testably verifiable. It will also be easier for an engineer to later make a design to meet this requirement as they now know how much energy the system should store.

Step 11: As a final check make sure that you've written your functional requirements "Functionally, not structurally." This will become natural in time, but when you are starting out, below are a few comparisons that may be helpful.

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 you measure your performance	The actual solution's performance
Anything that can meet this description is a valid solution	A very specific solution

Step 11 is complete when you have checked all your requirements to make sure they are written functionally, not structurally to help ensure you are not artificially constrained as to what your System must structurally be but are open to creatively think about possible solutions that could meet your challenge's functional needs.

The use case behavioral diagram (UCBD) gives you some idea as to how your System has to do all of these things together. There are other techniques, such as creating Functional Flow Block Diagrams (FFBD), IDEF-0, and Operational Description Templates (ODT), some of which can also help you to tie use cases together, break your System into subsystems, and identify key interfaces between them. Also, see the

Interface Tracking Guide, because if there's one place systems fail, its usually at the interfaces. These are outside the scope of this guide and we'd recommend that you start with an FFBD if you're interested in exploring this further. But either way, you are in a far better place to discuss and work on your design as a team towards a better defined System solution for your challenge.