# Key rules of our architecture

## Growing your application.

We start from a green field. We start with a minimal harness to run our first test successfully. What I mean with that is that although our test needs to fail first we have to foresee a testing framework to run our test in. We need to be able to run our test and some other non-functional requirements to work fast and get the necessary feedback. Every time a new test is introduced and to make this test succeed we foresee that the absolute minimum is extra included.

The second requirement is that the implementation of each test

## Technology independence

Technologies, libraries and other functionalities serve the tests and not the other way around. When a test requires for example that data will be stored in a database we introduce a database connectivity library (in our case it will be Entity Framework). We follow the advice and introduction steps that has been given in this document to implement it and go as quickly as possible to make our test succeed. We keep the principles in mind and we develop against them.

## Tests first

Never introduce more code then is needed to either get a test to a green state. Do the introduction steps of a certain technology, or during refactoring to tighten already written code.

It is not easy to not introduce code because you know it will have to be implemented in the near future. Keep it in mind and introduce it when a test requires it.

# Analysis and Design

We use Sparx Enterprise Architect version 12 as our design tool.

## The design process

We have a particular design process that we strictly need to follow when we have a new requirement that needs to be implemented.

Requirements

With the project of EAS we always start from a requirement. If a requirement is too complex or too large to handle we need to split it up in sub-requirements. You have to be careful you do not already go too much into detail and define use cases instead of sub-requirements. There is no clear definition when it is a sub-requirement or a use case but when there is an external system (an actor) that could call the definition we know that it is probably a use case.

*“If it is not clear enough to comprehend try to split it up in edible pieces.”*

This is an advice that works everywhere especially when you are working with a complexity as we have here in the requirements.

In some cases the requirements need to be decomposed and being studied in more detail with all stakeholders. For that we have the Archimate Business Layer Diagram where you can design the business processes being used.

Requirements

Archimate

Business Layer

Take notice that the business layer diagram is a complementary diagram and its sole purpose is to clear out certain ignorance.

When you have finished with your requirements and your business layer diagram when needed it is time to create the features and your use cases.

Requirements

Features

Use Cases

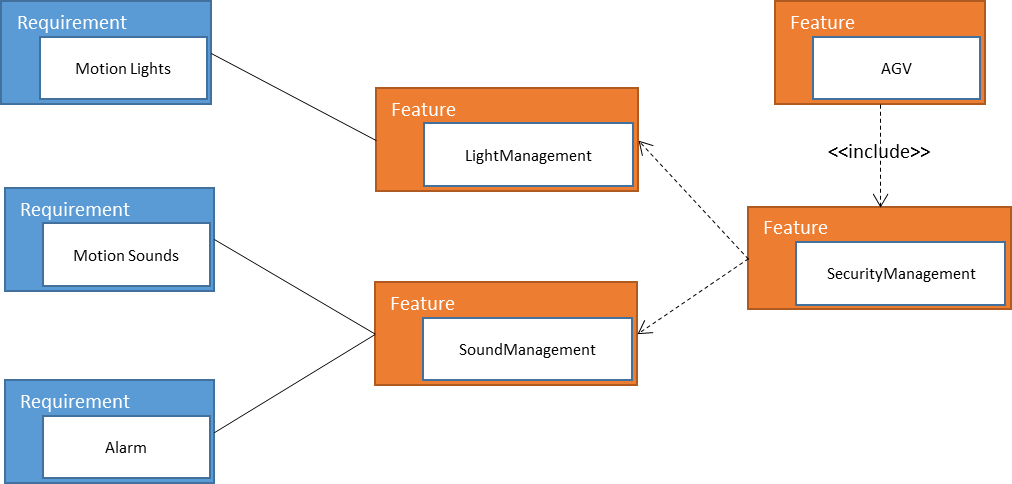
Steps in clear words

Requirements: What are the needs?  
Feature: Is an answer on the requirement. Composing the requirements in features.  
Use case: Happy flow to complete the requirement.

## Features

Several requirements can be composed together in features and can lead to effective groups. If we look at it from a technical view we can see that some requirements can lead to build a certain service, domain, screen or even higher to a separate application. It is possible that one requirement can be part of more than one composition or none.

An example says more than a theoretic description and so you get three requirements that we define the features for and relate them to already existing features.



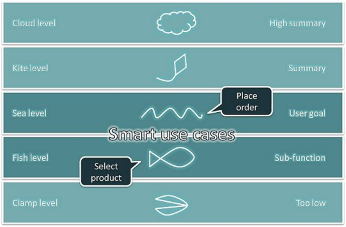
We have three requirements (shortened in their description for demonstration purposes) motion lights, motion sounds and alarm. We draw a diagram to define the features like the example in the picture above (we keep using the requirements diagram for that [[1]](#footnote-1)). How they need to work and what are the consequences will be described in the use cases but here we are going to define the features. As we do not define our hardware in the features we go to the features that are nearest to the hardware and those are the drivers or the managers. In case of the requirement *Motion Light* we define the feature *Light Management*. The same thing can we do for the requirement *Motion Sounds* we define the feature *Sound Management*. As the requirement *Alarm* do use sound we can say that this requirement also can be housed into the *Sound Management* features. Both features are a part of the security measurements of the product and so the feature *Security Management* is the owner of the other two features. To be complete in this example you can say that the *Security Management* feature has no purpose to live on its own (maybe it does as it could be something that is sold separately as a service for example).

The features are mostly being created to understand how the requirements are situated in the product. They maybe influence the use cases and other supporting diagrams but in an indirect way as a sole purpose of understanding for the designer or designers who are working with it.

The main idea behind defining features is to write documentation starting from them. You know what requirements are in them an instead of asking yourself what is needed you can write here what do we have. You can only start to write the documentation when the containing requirements of that feature are worked out (worked out = test cases are written and supported by application and technology layer diagram).

## Use cases

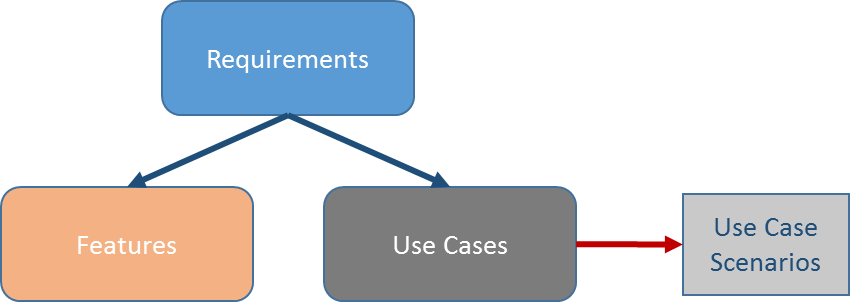
A use case describes the way your system behaves to meet a requirement. And that is not easy because there is not one solution for every requirement. There are a lot of interpretations. I am going to try to help you out with telling you to look at the requirement and see how you can divide it in logical steps by describing the requirement in one sentence. If you need more than one sentence your requirement is probably too big or you are going to deep into detail. The best thing you can do if you have the chance is discuss the requirement with someone else and play with the describing sentence. To help you with if you are going to deep into detail we say you need to stay on the sea level. The following drawing shows what I mean.



Definition

A use case is something that provides some measurable result to the user or an external system.

You can describe a correct sea level use case in the following sentence “The customer places an order”. The customer is the actor in this case and we talk about how to define your actor later on. In this case the use case is “places an order” or more general “place order”. We try to define our use cases as general as possible as you do not know how many more actors or other influences there can be. In this case “select product” or “select quantity” or “validate availability” are fish level definitions and that is to detailed. Because if we want to make a sentence out of that it sounds like we read off a list and probably need more than one sentence. The fish level we keep to describe the use case scenario’s[[2]](#footnote-2). When you have defined a use case you define your scenarios that are the steps you need to take to get to that use case.



Besides the happy path it is best to define the possible alternate paths that the use case can go through. And if there are exceptions that are important that can occur and what steps there need to be taken if so.

Be careful with exception paths. Field validation or going too much into detail is absolutely not necessary. Those possible failures we catch on the level of our test cases. You may define them but the risk can be that it changes to often and gets stale. The difference between an alternate or exception path in a use case scenario:

* An alternate flow is a step or a sequence of steps that achieves the use case’s goal following different steps than described in the basic (happy) path. But the goal is achieved finally.
* An exception flow is anything that leads to NOT achieving the use case’s goal. Anything that leads to NOT achieving the use case’s goal is an Exception.

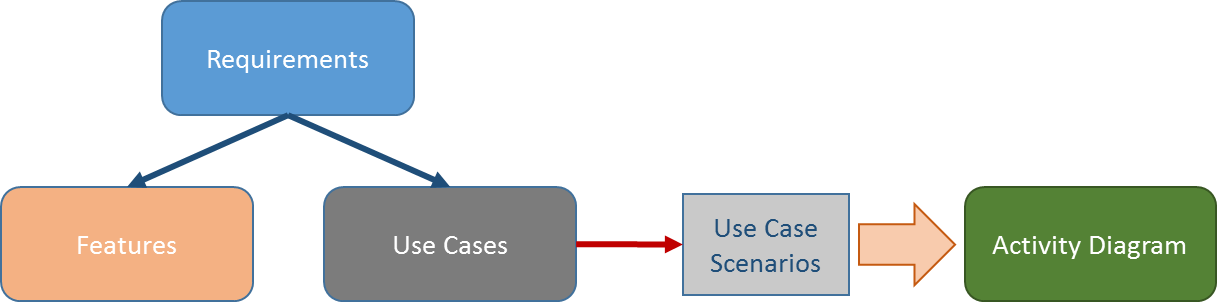
If you are designing an alternate path and it shows that you need another alternate path in there it shows that you didn’t define your use case on the sea level but higher (> kite level). Best to look at your use cases again and try to define them in more detail.

Ask yourself two questions:

1. How much detail is necessary to understand the use case?
2. What is the most likely scenario of the use case?

You are not writing pseudocode of some sort. And leave your technical head of. Try to think in steps.

When your first attempt to write down the scenario steps in the use case it is time to look at it from another perspective that give you a different understanding and shows you hopefully insight on the use case.



We generate an activity diagram out of the use case scenarios. We modify that activity diagram with the insights we got by understanding the use case from that perspective. Keep it on the fish level and leave out the detail. There are other places where you can go with the detail. It needs to be a clear view and understanding of your use case. Keep in mind that the implementer or your colleague need to have an instant clear view on what is the intention of the use case.

When you are confident that the activity diagram together with the scenarios are explaining what the use case is we define in the application layer diagram how it situates in the software architecture.

1. See the step-by-step guide for EA how exactly this example is implemented. [↑](#footnote-ref-1)
2. See the step-by-step guide for EA for more on use case scenarios. [↑](#footnote-ref-2)