# 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.

# Quality attributes

Transactional

Technology independence

Configuration system

Real-time computing

Portability

Extensibility

Recoverability

Immutable

# 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

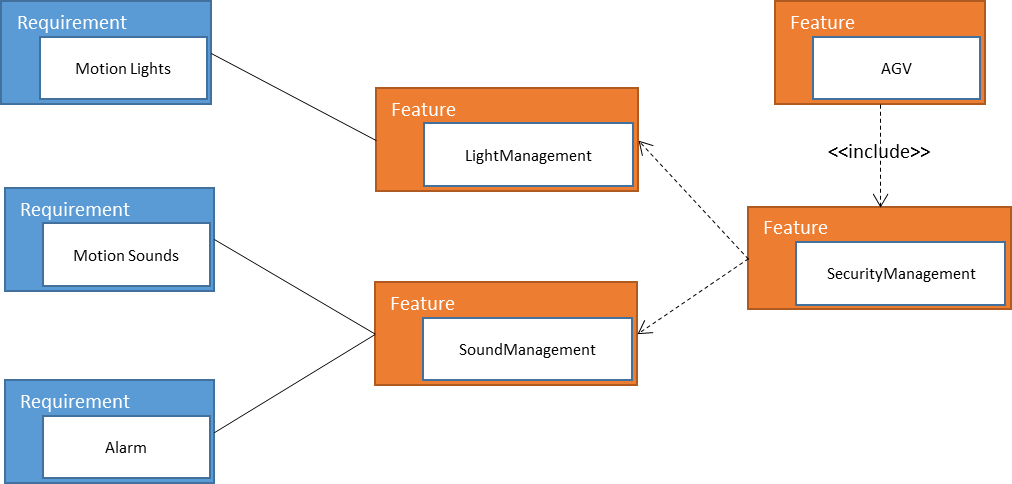
Steps in clear words

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.



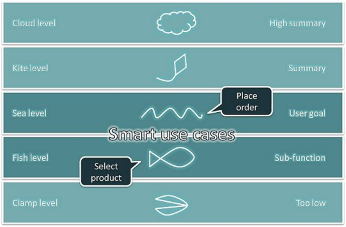
We have three requirements (shortened in their description for demonstration purposes) motion lights, motion sounds and alarm. We draw a diagram to define the features like the example in the picture above (we keep using the requirements diagram for that [[1]](#footnote-1)). How they need to work and 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

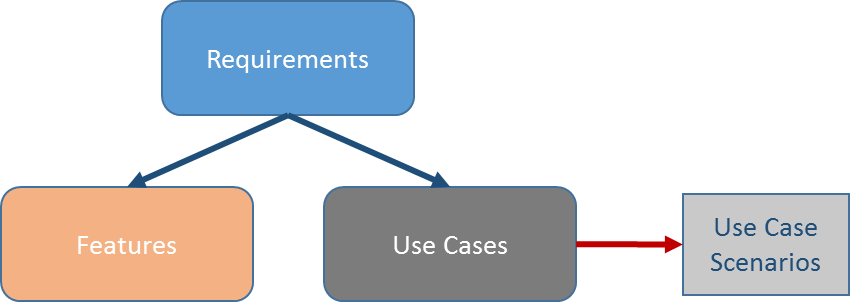
A use case describes the way your system behaves to meet a requirement. And that is not easy because there is not one solution for every requirement. There are a lot of interpretations. I am going to try to help you out with telling you to look at the requirement and see how you can divide it in logical 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.

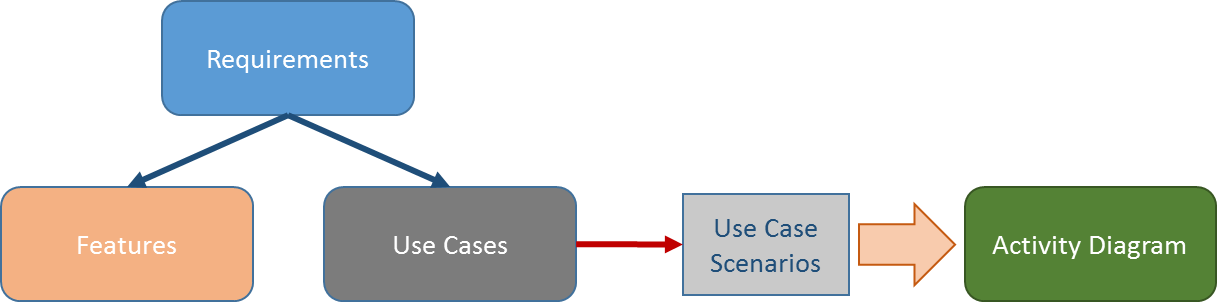
If you are designing an alternate path and it shows that you need another alternate path in there it shows that you didn’t define your use case on the sea level but higher (> kite level). Best to look at your use cases again and try to define them in more detail.

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?

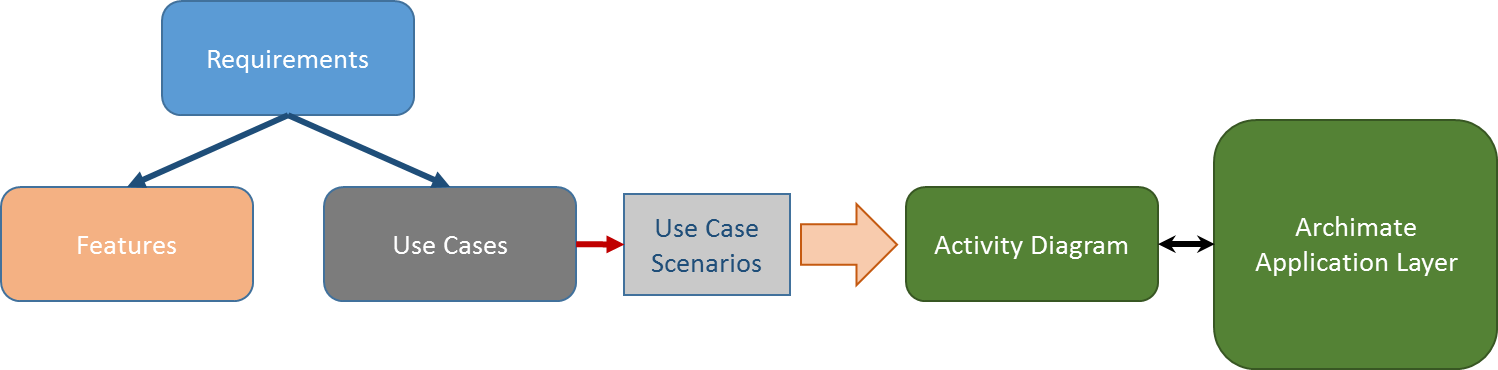
You are not writing pseudocode of some sort. And leave your technical head of. Try to think in steps.

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.

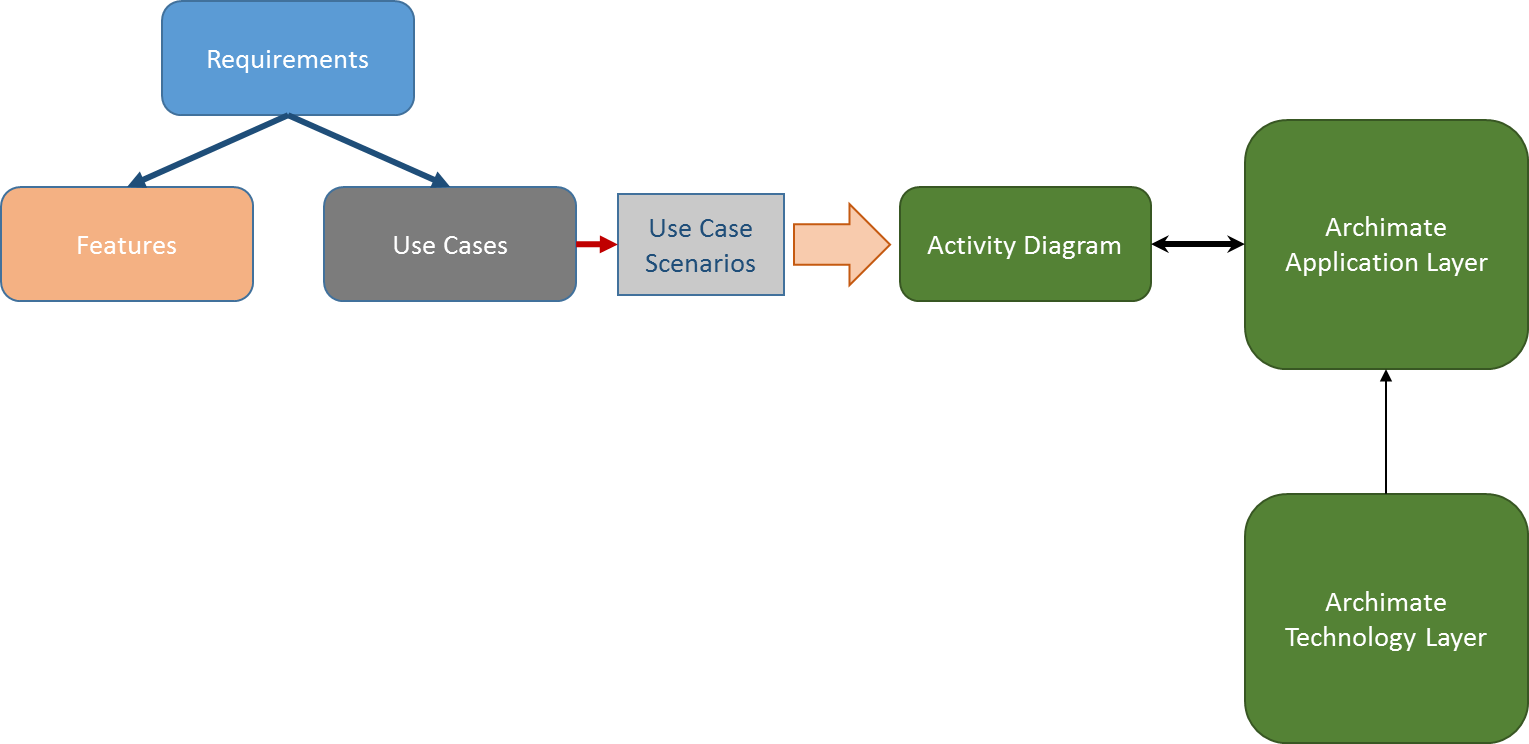


We generate an activity diagram out of the use case scenarios. We modify that activity diagram with the insights we got by understanding the use case from that perspective. Keep it on the fish level and leave out the detail. There are other places where you can go with the detail. It needs to be a clear 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 will fit in the software architecture. We use for that the Archimate application layer diagram.



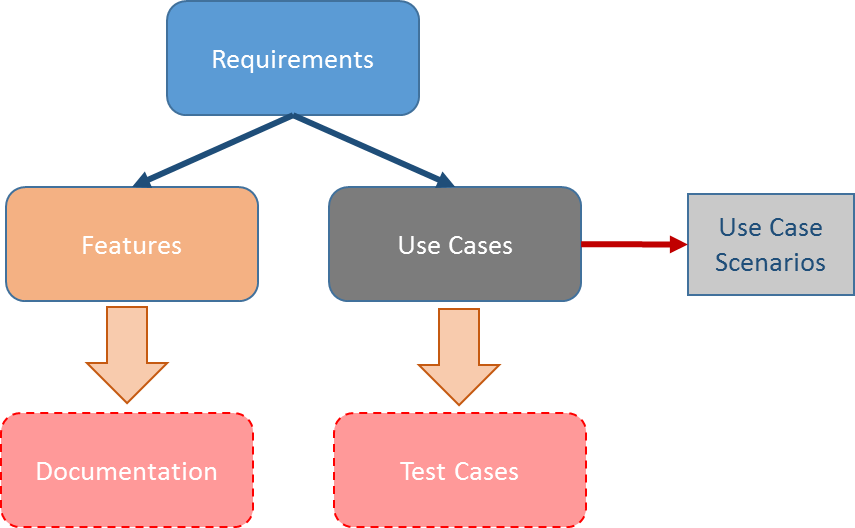
As we are working always towards a walking skeleton we early on need to know how the application components need to be installed to work correctly and represent how the requirements was intended. Before doing the necessary DevOps to get to the walking skeleton we need to design how it needs to be deployed and for that we use the Archimate technology layer diagram. We design it according to how the application layer diagram is been composed.



It could be that you are not being able to design your technology layer diagram because the requirement does not offer enough information to do it or relies on others. And so you need to design it in a different order. That should be no problem as this diagram does not influence the other steps in the design process (you can call the technology layer diagram a stub diagram).

Stub diagram is a diagram where no other diagrams depend on.

So the next step as we finished the technology diagram is to generate the test cases out of the use cases. Although if needed and in some cases (when you are working on complex material) it is needed that you generate them earlier when the use case scenarios are finished.   
The test scenarios are automatically generated as well but will probably not suffice. It is advised to write extra manual test scenarios so your test coverage on the use case is more tight and helps with ruling out any exceptions[[3]](#footnote-3).



When you have finished generating and writing the test cases you can distill the documentation out of the features in accordance with the information you can get from the requirements.

# Test Driven Development

As it is one of the key values we need to take a lot of attention to it. It is the key value where it all starts when you have designed a requirement to its detail. To write and run tests we need to have a test harness that provide us to be able to that.

## Test harness

The test harness contains the tools to write, run and get the necessary feedback for our tests defined in the test cases of our requirement design. As we do TDD we follow the rule of Red, Green, Refactor. Where Red means a failing test to commence with and so Green is next progression where we try to succeed our test with the least possible steps. To finish off the test we refactor the according code so it follows the principles and architecture we defined (see the walking skeleton).

### Test framework Xunit

To write our tests we need a framework that provides the opportunity to write tests. The framework has all possible assertions to fail the test. The framework needs to have a runner so we could run our test and get the feedback we need to have (feedback as in failing or succeeding the test).   
The Xunit framework has the fastest runner and foresees a minimal package of instruments to write your tests. The learning curve is because of the minimal package very low and every developer should be able to start with it immediately. Although it has a minimum package it contains all the patterns[[4]](#footnote-4) we need to be able all sorts of tests (unit, integration, system, …).

Xunit exists out of two nuget packages you need to install in your test projects to be able to run your tests. In the package manager console you enter the following commands:

PM> Install-Package Xunit

PM> Install-Package Xunit.runner.visualstudio

### Refactoring tool Resharper

With the test framework Xunit installed we are able to run our tests and succeed in the Red and Green step. We can do the refactoring of our code perfectly manual. But if we have a tool that can guide us and do the refactoring work very quick and easy that would be helpful and saves us a lot of time. Resharper 9 from Jetbrains is the perfect tool[[5]](#footnote-5) for that and is sitting in our default tool belt.

### Automated test feedback tool NCrunch

As a developer getting feedback if we are on the right coding track or not is important. If we can do that without losing a lot of time or by not have to do a lot of steps that can make the implementation toward succeeding tests it would make the resulting code be more reliable and the delivery time shorter.   
With NCrunch we have an automated test feedback tool that gives us immediate feedback of the tests we are working on. If a test we changed fails or not. If my by changing some code we didn’t break somewhere a test. If you get immediate and accurate feedback you do not forget to run the tests and you know what is happening.   
If you look at the tool and did not use it yet you can think what is the extra asset of a tool like this? After working with it for your first tests you immediately will find out that this is a tool that is indispensable and must hang on our tool belt.

### Mocking library FakeItEasy

In some cases you need results out of dependent objects of the tested class and method. You can make your own test class on the dependent interface. That is ok if it happens only a few times that you need to write such a test class but when the test environment gets big it becomes a lot of work. When you write a test class like that you are also responsible to maintain it. That can be prone for mistakes and errors.   
For that there exists mocking libraries and FakeItEasy is one of the modern .NET ones.

### Version control with Git

Every piece of code that has been written must be backed with a test and so you can assume that tests are the starting point of our code. That means that tests are as important as the eventual code of the implementation. We do not lose them and want to keep and maintain them. So version control instead what usual being done by linking it to the walking skeleton we say that version control is part of the test harness and we drive it from the test perspective.

There exists a lot of different kind of version control systems that are distributed but none is as distributed and simple as Git. By using a disconnected version control we take care of a developer blocking another developer. It will not happen and by using a flow that we call “Pull request Workflow” we can work truly distributed over the two sites in USA and Belgium without having connection issues.

TODO: Work out an example with visualstudio.com on how to connect and make a pull request.

## Start guide

TODO: Write a step-by-step to start from the test case to a refactored implementation.

## Walking skeleton

The test harness will evolve to a walking skeleton as we called it. The skeleton is an implementation of the thinnest possible slice of real functionality that we can automatically build, test and deploy end-to-end. You can say that the harness not actually evolves because it still exists during the implementation of the tests but that is not important.

A walking skeleton is an implementation of the thinnest possible slice of real functionality (no tests although they are as necessary) that we can automatically build, deploy, and test end-to-end. It should include just enough of the automation, the major components, and communication mechanism to allow us to start working on the first requirement.

Besides the walking skeleton it is absolutely prohibited to implement more then what makes your test pass. It is hard to not get to be lured in to write more than intended to get the test passed. The thing is when you write more you get the chance it is not test covered.

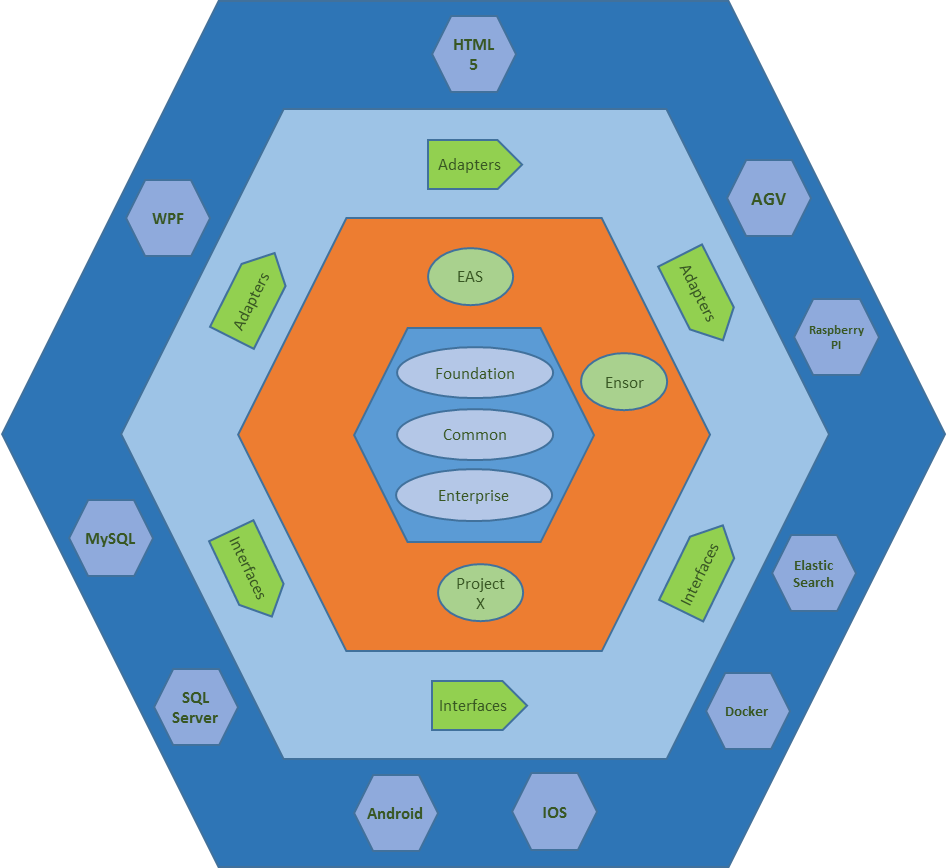
We present now parts of the skeleton that is the software architecture. You can say that already foreseen too much but I do not agree with that. You have to make those decisions because they point you in a certain direction or need to be part of the foundation that are hard to change.   
We call software architecture, the components that are hard to change later. The basic principles are not only a foundation where other components rely on but you can consider them also as the guides to work to the goal.

### Hexagonal architecture

As we have defined our quality attributes we need to find an architecture that meets those attributes. It turns out that the classic way of using the multitier architecture could not be done. We could not meet the key values and the attributes with that architecture.   
The technology independence is not guaranteed by a multitier environment and could lead to a tight-coupling of important parts.

So I had to look further to a technique where we can guarantee that our business logic could stand the ravages of time. Reading an article of Robert Martin[[6]](#footnote-6) I found the right solution, the hexagonal architecture. Set on the architecture map by Alistair Cockburn[[7]](#footnote-7)

An hexagonal architecture is an applicative-architecture style that helps us to focus on our business goals without being tied or jeopardized by our technical frameworks or other technologies. There is no front or backend explicitly. You have the inside and the outside. By using the dependency inversion principle which states that high-level modules should not depend on low-level modules. Both should depend on abstractions, you can easily infer that this model dictates that you can only point inwards the hexagon. Interactions between the inside and the outside are done by ports and adapters (interfaces).



The inner hexagonal will encapsulate the enterprise-wide business rules and entities. Every component in the hexagonal could be used by many different applications in the enterprise.

The next hexagonal layer is where the application specific components reside. In our case it all code that is the business logic and technology independent for EAS or Ensor.   
Those two layers are the inner circle of the architecture and it is important that revisions and refactoring takes place to keep those hexagonal layers clean.

The outer hexagonal layer that we call the outer circle is the one that holds the outer points (the frontend and the backend from the multitier architecture for example). As a multitier is more intended for one frontend and one backend the hexagonal architecture can adapt to any technology component in the outer hexagonal layer. The outer layer is detail and contains no logic whatsoever

The interface adapters layer in between the outer and inner layer you can call the port between them. Drivers between components of both sides. The enterprise kernel can never be called from the outer circle. This is to prevent that the code becomes unmaintainable and stale.

And when you develop in this architecture you know that the SOLID principles are your best friend. They have everything inside to make the hexagonal architecture as dynamic as it needs to be. When I look at this architecture I always compare it with an operating system that is a more comprehensible hexagonal architecture. You have also the kernel, the applications, the adapters to what you see or what you want to output.

In a multitier architecture you can clearly separate the different layers in separate namespaces or projects and work from there. In an hexagonal architecture you do not have a clear distinction like that and have to work with more than only namespaces and projects.

### Scalable, distributed real-time transaction processing with Akka.NET

### Basic principles

#### SOLID

* Single Responsibility principle (SRP): A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class, There should never be more than one reason for a class to change. When writing code, keep in mind that classes should be as simple as possible and focus on one main core task. ).
* Open/Closed principle (OCP): Software entities should be open for extension, but closed for modification. It means that an existing class should be extensible and usable as the foundation for building other related functionality. But to implement other related functionality, you should not change the existing code which, subsequently, remains closed for modification. Composition, interfaces, and generics. For example, if you program a class to log its activity using a generic logging interface, you automatically enable that class to use any logger that exposes the interface. If you use composition, you can build up new functionality on top of existing components without touching them.
* Liskov substitution principle (LSP): Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program (design by contract). Composition is a safer way to code classes. Subclasses should be substitutable for their base classes. The essence of this principle is that a derived class can't restrict conditions under which a base class executes. A derived class should require no more and provide no less than its parent. Whenever you define a virtual method, you should make sure you're not calling any private member from within it.
* Interface segregation principle (ISP): Many client-specific interfaces are better than one general interface. Clients should not be forced to depend upon interfaces that they do not use.
* Dependency inversion principle (DIP): Depend upon Abstractions. Do not depend upon concretions. We know it more as dependency injection and probably used it in the past more as a way to abstract for testing purposes. With the hexagonal architecture we are going to use it also for implementing the different layers with each other and abstract parts so we can approach it from different perspectives.

#### YAGNI

YAGNI or You aren’t gonna need it is a principle that states you should not add functionality until deemed necessary. This rule hangs close with our test harness and walking skeleton. Only use what you need to make the test green and that’s it.

#### DRY

DRY or don’t repeat yourself is one of the core principles that you never ever can’t repeat code. This is a principle that you always have to take into account when you are refactoring your code. Get rid of the duplicate code.

Every piece of knowledge must have a single, unambiguous, authorative representation within a system.

If I am the reviewer duplicate code would lead to not accept your pull request and so you need to fix it and reinitiate it.

#### Law of Demeter

Objects make decisions based only on internal info or trigger a message. Avoid navigating to other objects.

This means concretely in technical terms. Your method can call other methods in its class directly. Your method can call methods on its own fields directly (but not on the fields fields). When your method takes parameters, your method can call methods on those parameters directly. When your method creates local objects, that method can call methods on the local objects. What you cannot do is, call a chain of messages A.GetB().GetC().DoSomething() in some class other than A’s class.

Another explanation is that a method m of an object O may only invoke methods of the following kinds of objects.

1. O itself.
2. m’s parameters.
3. Any objects instantiated within m.
4. O’s direct component objects.

An object should avoid invoking methods of a member object returned by another method.

It is important that you follow this law. You will get into trouble to in your test driven progress if you go for example into those method chains. It is not easy to test that and we have to prevent to mock too much (although FakeItEasy is so fun to work with you still have to prevent it too not to overuse it for maintainability sake).

TODO : Examples in C# and C++.

#### Event Sourcing

The fundamental idea of Event Sourcing[[8]](#footnote-8) is that ensuring every change to the state of an application object is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application object itself.

A common example of event sourcing is version control systems. We introduce it in our product for several reasons, service maintenance, error handling, transaction and mission critical actions. You need to be able to reproduce exactly what you have done and a log entry or entries will not tell you always the very truth even if it intended too. Event sourcing has zero chance to be misinterpreted and that is why I consider as an architect that this is a must for our product.   
The good news is that our actor-based framework has that feature by default and so it is only our task to handle it properly and instrument it for its purpose.

### Architectural patterns

#### CQRS

#### Loopback

### Premature technology choices

#### Akka.net

#### Entity Framework

We are using the code first approach of the Entity Framework. I go for this ORM because it has an easy learning curve and is friendly for working in a test driven way. Microsoft gives its full support and even went for a full rewrite in version 7 to ensure it is ready for the future.   
Our hexagonal architecture welcomes an ORM library like Entity Framework because it is ready into its core for dependency inversion. It gives us the opportunity to go to the bare metal if needed when performance-critical queries need to be written.   
The coding style is fluent so it is easy on maintenance and gives a clear result when problems should occur.

Entity Framework 7[[9]](#footnote-9) is designed to “New Platforms, New Data Stores”. And this with support for non-relational data stores and in-memory data. Support for phones and tablets. Support for Linux and OSX. And another asset that can make the difference is that it is opensource.

#### NLog

### Start guide

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)
3. Look in the step-by-step guide for EA how you need to write the manual test scenarios. [↑](#footnote-ref-3)
4. Meszaros, Gerard (2007) *xUnit Test Patterns*, Pearson Education, Inc./Addison Wesley [↑](#footnote-ref-4)
5. Our Resharper 9 manual helps to learn all details about this tool. [↑](#footnote-ref-5)
6. http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html [↑](#footnote-ref-6)
7. http://alistair.cockburn.us/Hexagonal+architecture [↑](#footnote-ref-7)
8. http://martinfowler.com/eaaDev/EventSourcing.html [↑](#footnote-ref-8)
9. https://github.com/aspnet/EntityFramework/wiki/What-is-EF7-all-about [↑](#footnote-ref-9)