Divide & Impera - or:
How to split requirements for a large product into digestible parts

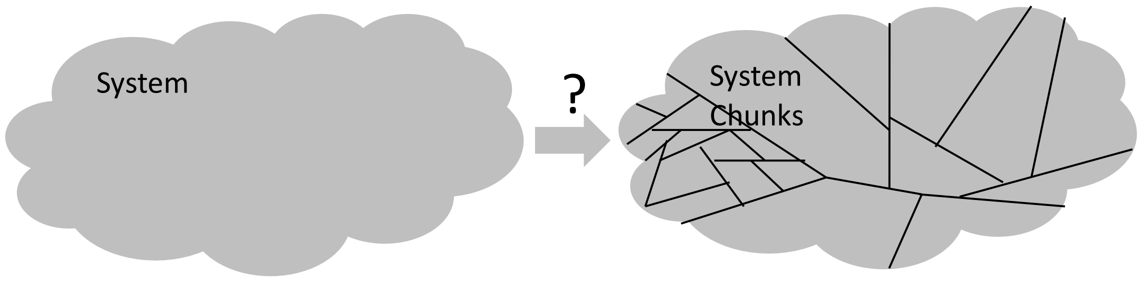
Peter Hruschka, May 2017

In this paper we discuss strategies and techniques for splitting a big problem into smaller problems. This divide and conquer strategy is well known: You want to recursively break down a problem into several subproblems until they are simple enough to be solved. In agile IT-terms this means that you have to continue dividing until the parts are small enough to be implemented within a single iteration.

For such parts we have used many names in the past: we talked about modules, we have called such parts subsystems or components. Then we learned to decompose a system into use cases and with the advent of agile methods terms like themes, epics, features, and user stories became fashionable.

Splitting problems in the large

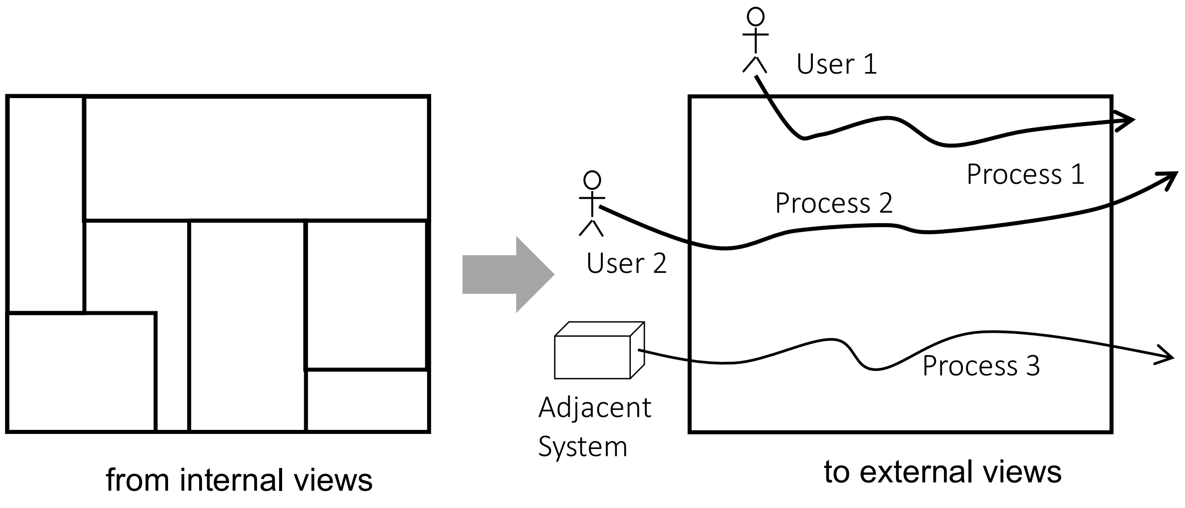
There are many strategies and tactics you can use to decompose a whole system into parts. Let's have a more systematic view at criteria that can be applied to split a big problem into smaller chunks.



The most obvious criterium is to identify **logically cohesive functions**, i.e. groups of functions that together perform some related work. Such groups of functions are often called **epics** or **themes** or **features** today. In an e-commerce application that would e.g. be "managing the product catalogue", "searching and ordering articles", "shipping and invoicing". In a home automation system, you might identify "configuring of the system", "controlling devices", "adjusting parameters", etc. Many of them may still be large and complex, but at least you identified some smaller chunks that together make up the whole system.

Are there other criteria than searching for logical functions? Of course: you could split by **organizational boundaries** (support for production, support for sales, support for marketing, …). Or you could split **geographically**: Things we need in Italy, in Germany or in France. Another possibility is splitting by hardware or software-architecture: things needed on the client-side and things for the server-side.
Or your group functionality around objects: everything you need to do with employees, with contracts, with orders, …

And the most obvious split - if you have an existing system - is to use the **structure of the existing system**, i.e. split the that people before you have chosen in order to build that system.

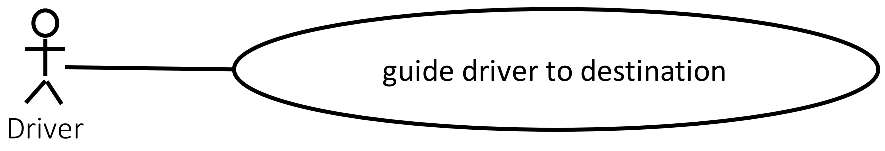


Note, that while all these criteria may be useful, they suffer from one common restriction. They are all based on **internal views** of the system: they result in the systems internal functions, its internal organizational structure, its hardware or software structure, its objects or its historical structure.

Shift your decomposition strategy from internal structures to externally triggered structures

Many methods over the last 40 years suggested to shift your view. Why don't you start your efforts to decompose a system on the outside of the system: in its environment or in its context? Try to find triggers in the real world (outside your system) that want to be serviced by the system. This frees you from looking into the internal structure and brings you to and **external view**, i.e. to **system processes** that are triggered because of external needs.

The first ones to suggest that were Steve McMenamin and John Palmer in the mid 80s: In their book "Essential Systems Analysis" they suggest finding events in the real world or time events that trigger system processes. They used the term "event-oriented decomposition" to find business processes or system processes. Professor Scheer in Germany picked up on that idea and suggested EPCs (event process chains) as decomposition criterium. The - in 1992 - Ivar Jacobson called such externally triggered processes "use-cases" and suggested a notation consisting of stick-persons for the external triggers and ellipses for the (internal) processes, i.e. the use cases.
With the advent of agile methods user stories became more popular. Mike Cohn replaced the graphical notation of Ivar Jacobson with the text pattern "As a <role> I want <functionality> so that <advantage>.
Consider the example of a navigation system. It is really equivalent to either draw a use case or write a story as shown in the following. They are identical in their meaning and are based on the same strategy of starting with external triggers.



As a driver I want to be guided to a destination so that I can concentrate on the traffic

So, it is more or less a matter of taste whether you prefer a graphical use case diagram or a list of (textual) user stories.

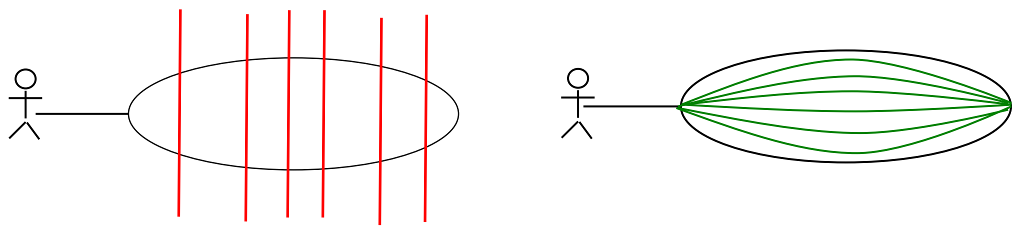
Let me summarize the discussion about structuring requirements in the large: Independent of notation a decomposition of a big problem into externally triggered process is the most neutral way to achieve this breakdown. It is NOT based on any internal structure of the systems (not its functions, not its subsystems, not its objects, .,,) but purely on needs from the environment.
This is not to say that the other decomposition strategies mentioned above don't work, but only a suggestion to not get stuck in internal structural discussions.
Splitting in the small

As you can see in the simple example of the navigation system above this process (independent of the notation you prefer) still seems to be too complex to be implemented in one iteration. You definitely need to break it down into smaller chunks. Lawrence (https://agileforall.com/resources/how-to-split-a-user-story/) suggested many way to tackle bigger chunks and divide them into stories that are small enough for one iteration.

While classical methods suggested mainly to understand a complex process better by looking at its sequence of steps, we are now facing an additional problem in the agile world: we want to create value in every iteration, i.e. come up with potentially shippable product increments. It is not good enough to just decompose a big process (a use case or a large story) for understanding it better. After decomposition each of the part still should deliver some value to the user independent of the other parts.

This inhibits technical splits like splitting into its technical layers like user interface, domain functionality and persistence layer, since implementing the persistence layer first (i.e. creating some database tables) does not deliver any business value until you can fill them and display them. The same is often true for pure business process decomposition into its steps: implementing just the single step 7 of a complex process may not deliver any value as long as predecessor and successor steps are not there.

(Just an interesting historical side remark about terminology wars between methodologists: Mike Cohn complained that user stories are not use cases simply by using the argument: They may be too big for implementing them in one iteration. Ivar Jacobson accepted that argument about size, but stroke back by claiming the "use cases slices" are a much better way of thinking then working with small enough user stories, since use case slices deliver end-to-end-value.)



The suggestion is that you try slicing into end-to-end processes and not cut a business process into its steps. Based on that principle strategy I try to simplify the complex suggestion of Lawrence mentioned above by suggesting that you mainly have three ways to shrink such slices to a size that fits into one iteration:

1. You can leave out alternatives (for example first go for the normal flow, adding exceptional cases later on - or implementing the process in one technology first and adding others later)

   For the navigation system above, you could implement the selection of the destination by entering city, street and number first, leaving picking from favorites, or speech input for later.

1. You can leave out options (for example leaving out things that are not absolutely necessary to be implemented in an early release).

   In the navigation example you can first calculate the quickest route only, leaving other options like shortest route, scenic route, routes avoiding toll streets, … for later releases.

2. You can leave out steps for which you have manual work-arounds in early releases until you find the time to automate them.

   In the example above, you may still have to listen to traffic messages in the radio and consider actions yourself while later on traffic messages may automatically be included in your routing algorithms.

Summary
Try to avoid methodological terminology wars. A process-oriented decomposition is a good strategy to divide and conquer. Do not fight between calling is a use case or a user story (or by any other name).
In order to create quick potentially shippable product increments you face the problem to not only understand complex processes but also slice them in a way that the slices still deliver business value. The suggestion in the paper might help you to tackle complex problems and create quick wins.