Capturing Agile Requirements by Example (CARE)

Syllabus

Version: 1.0

Released: August 2022

Copyright Notice

Thank you to the main author

- Kaspar van Dam

Thank you to the co-authors

- Piet de Roo
- Patrick Duisters
- Mark de Munnik

Thank you to the AU review committee

- A special thanks to Wim Decoutere and Carlo van Driel who's detailed comments really helped for form the final version of this syllabus.

Revision History

Version	Date	Remarks
0.1	March 2021	Initial Beta release
0.2	April 2021	Draft to proposal
0.3	October 2021	Draft ready for review
0.4	December 2021	Editorial review
0.5	April 2022	Processing review comments and final edits.
1.0	August 2022	Final version of the syllabus

Table of Content

Business Outcomes

Business objects (BOs) are a brief statement of what you are expected to have learned after the training.

BO-1	Understand the principles of CARE and BDD in general.
BO-2	Understand the different roles involved in the different stages of CARE.
BO-3	Have insight in the required mindset to make CARE successful.
BO-4	Be able to implement Impact Mapping and Impact Mapping sessions.
BO-5	Be able to implement Event Storming and Event Storming sessions.
BO-6	Be able to implement Example Mapping and Example Mapping sessions.
BO-7	Be able to use Specification with Examples to create a shared understanding.
BO-8	Understand the principles of (Acceptance) Test Driven Development ((A)TDD).
BO-9	Understand how (A)TDD can be used to turn Specification with Examples into Executable Specification.
BO-10	Use your knowledge of CARE to successfully implement CARE when creating software.

Learning Objectives/Cognitive Levels of Knowledge

Learning objectives (LOs) are brief statements that describe what you are expected to know after studying each chapter. The LOs are defined based on Bloom's modified taxonomy as follows:

Definitions	**K1 Remembering**	**K2 Understanding**	**K3 Applying**
Bloom´s definition	Exhibit memory of previously learned material by recalling facts, terms, basic concepts, and answers.	Demonstrate understanding of facts and ideas by organizing, comparing, translating, interpreting, giving descriptions, and stating main ideas.	Solve problems to new situations by applying acquired knowledge, facts, techniques, and rules in a different way.
Verbs (examples)	Remember Recall Choose Define Find Match Relate Select	Summarize Generalize Classify Compare Contrast Demonstrate Interpret Rephrase	Implement Execute Use Apply Plan

For more details of Bloom's taxonomy please, refer to (Anderson, Airasian, & Krathwohl, 2001) or (Anderson L. K., 2022).

Normative vs. informative

Each chapter in this syllabus starts with keywords and learning objectives. These are the things that are examinable and need to be learned in order to pass the certification exam, according to the levels of knowledge as described before. All other information in this syllabus is informative and elaborates on the learning objectives.

Hands-on Objectives

Hands-on Objectives (Hos) are brief statements that describe what you are expected to perform or execute to understand the practical aspect of learning. The Hos are defined as follows:

- HO-0: Live view of an exercise or recorded video.
- HO-1: Guided exercise. The trainees follow the sequence of steps performed by the trainer.
- HO-2: Exercise with hints. Exercise to be solved by the trainee, utilizing hints provided by the trainer.
- HO-3: Unguided exercises without hints.

Prerequisites

Mandatory

- None

Recommended

- General Knowledge in Agile Software Development and the Agile Manifesto
- Basic knowledge in Requirements and Requirements Engineering

Introduction to CARE

Capturing Agile Requirements by Example (CARE) is all about requirements, but it is also about agility, about communication and collaboration. It is a mindset, a way of working and a tool.

*CARE combines the principles of Behaviour Driven Development (BDD)[1], Impact Mapping, Event Storming, Example Mapping and Specification by Example. It provides the means to practically implement these things and to help create software an end-user really **CARE**s about.*

[1] The term *Behaviour Driven Development* has its origin in Great-Britain, because of this, the word 'Behaviour' is written in British in this syllabus and not using the American spelling where it would be 'Behavior'.

Chapter 1 : Agile Requirements

The understanding of CARE begins with understanding what requirements are and also their position within an agile way of working.

Keywords

Requirements, Agile Requirements, Testing, Product Risk, Project Risk, Quality Risk, Agile, Manifesto, Known knowns, Unknown knowns, Known unknowns, Unknown unknowns, Agile Testing, Scrum

LO-1.1	K1	Define what a requirement is.
LO-1.2	K1	Recall what the different definitions of a requirement are.
LO-1.3	K1	Define what an agile requirement is.
LO-1.4	K2	Explain the differences between a 'regular' requirement and an agile requirement.
LO-1.5	K2	Summarize why requirements are important.
LO-1.6	K2	Demonstrate which purpose requirements serve within a (software) development project.

1.1 What are requirements?

LO-1.1		K1	Define what a requirement is.
LO-1.2		K1	Recall what the different definitions of a requirement are.

In software development, or more generally in product development, it is common practice to first find out and document what the new product is supposed to do before starting to build it. The statements that describe this are usually called 'requirements'. They often take the form of short sentences like 'The system will create a daily sales report,' or 'The database shall be backed up on a daily basis'.

Before defining any requirements, it makes sense to also define what a requirement is. There are many views on this. Ellen Gottesdiener states in her book 'The Software Requirements Memory Jogger' (Gottesdiener, 2005) that "Requirements are descriptions of the necessary and sufficient properties of a product that will satisfy the consumer's need", a rather formal definition. Donald Gause and Gerald Weinberg in their book 'Exploring Requirements' (Gause & Weinberg, 2011) do not strictly define what a requirement is, but just state that: 'Requirements are merely a guide. They are to be taken literally, but not too literally.' According to Suzanne and James Robertson (Robertson & Robertson, 2013), requirements are neither a description nor a guide, but they can exist without being documented: "… a requirement is something the product must do to support its owner's business, or a quality it must have to make it acceptable and attractive to the owner". Moreover, they make a distinction between functional requirements (things the product must do), non-functional requirements (qualities the product must have), and constraints (limitations or restrictions to consider).

These different interpretations of what a requirement is will inevitably lead to different expectations (Wiegers & Beatty, 2013) as they are the starting point for developing new products or new

processes and often are part of contracts between suppliers and customers and they will even change over time.

Currently, many parties, including IREB (the International Requirements Engineering Board) and IIBA (the International Institute of Business Analysis), make use of the definition described in the (superseded) IEEE Standard Glossary of Software Engineering IEEE 610.12-1990 (IEEE 610.12, 1990):

A requirement is either:

- A condition or capability needed by a stakeholder to solve a problem or achieve an objective.
- A condition or capability that must be met or possessed by a solution or solution component to satisfy a contract, standard, specification, or other formally imposed documents.
- A documented representation of a condition or capability as in (1) or (2).

The first part of the definition states that a description of the problem to be solved is a valid requirement ("I need the product to support …"). The second part states that, if the stakeholder knows exactly how the problem must be solved, this can also be considered a requirement (albeit a constraint). The latter type of requirements can also have its origin in laws and regulations. The third part indicates that the documented version, i.e., the requirement text, is also called a requirement, which implies that requirements can exist without having been documented at all.

In 2011, IEEE redefined the term 'requirement' more concisely as a 'statement which translates or expresses a need and its associated constraints and conditions' (ISO/IEC/IEEE 29148, 2011).

1.2 What are agile requirements?

LO-1.3	K1	Define what an agile requirement is.
LO-1.4	K2	Explain the differences between a 'regular' requirement and an agile requirement.

In principle, there are no "agile requirements" or non-agile requirements. Requirements are not different, it is rather how one deals with requirements in a given environment that differs. As stated in the previous paragraph, requirements may not be documented.

Epics and user stories should not be confused with requirements. Epics or user stories are packages of work being done in an agile way of working like Scrum. Part of such a package is the (definition of) the requirements.

The biggest difference between 'agile requirements' and 'traditional requirements' is that traditional requirements are assumed to be defined and recorded before the actual development or creation is being done (waterfall), while agile requirements may be defined upfront, but they can also be further detailed during agile development as part of a user story, while the developer and end user/customer work closely together and adapt the requirement to what is really needed. As such, at least when the user story is done, the product is developed, and the requirement is clear.

This implies that the requirements process in agile environments needs to be adapted to suit agile development. Focus on: What does the business need the most? What delivers the most business value and is prioritized? How do you state clearly to the developers what needs to be built, at the right time?

1.3 Why (agile) requirements are required

LO-1.5	K1	Recall why requirements are important.
LO-1.6	K2	Demonstrate which purpose requirements serve within a (software) development project.

(Agile) requirements serve as a basis to get a shared understanding of what it is that the business really wants, both implicit shared understanding and explicit shared understanding (for reference see: (International Requirements Engineering Board e.V. (IREB), 2020), pp. 13). When this shared understanding is reached, one can decide on how the software can fulfil these wishes and demands. Only then one should start thinking about implementation.

There is, however, a continuous debate about the need to capture agile requirements, since some claim that, after implementation, the developed product (and functions) show what the software is doing and there will be feedback from the users if it meets the actual wishes and demands. This debate is fed by different interpretations on the Agile Manifesto stating one should value working software over comprehensive documentation (Agile Manifesto, 2001).

However, the lifecycle of the product does not stop after (initial) development. For further reference, future development, training etc., it is highly recommended to consider the requirements as part of the 'documentation'. This enables insight in what the actual product is (intended) to do, to perform, and how to work with it. Typically, with complex products, e.g., with (a range of) internal and/or external interfaces, there is a need to know how to communicate with the system. Furthermore, in many industries, system documentation is required to get market approval, to be allowed to sell and use the product on the market.

Besides that, one can question if it is very effective to write software without clear requirements only to risk having end-users tell you that it is not doing what they want(ed) it to do. Certainly, in an agile context changes in the developed software may be small, and if things do not meet the wishes and demands of end-users the loss of work may not be immense because of that, but it still is time/energy wasted.

Chapter 2 : Behaviour Driven Development (BDD): The Basics

In this chapter, we will define what BDD is and elaborate on the theoretical basis of how to use it.

Keywords

Behaviour Driven Development (BDD), Test Driven Development (TDD), Acceptance Test Driven Development (ATDD), Domain-Driven Design (DDD), Extreme Programming (XP), Agile Manifesto, Impact Mapping, Event Storming, Example Mapping, Specification by Example, Living Documentation, Three Amigos

LO-2.1	K2	Summarize what Behaviour Driven Development (BDD) is.
LO-2.2	K1	Recall the influence of Test Driven Development on the origin of Behaviour Driven Development.
LO-2.3	K1	Recall the influence of Acceptance Test Driven Development on the origin of Behaviour Driven Development.
LO-2.4	K1	Recall the influence of Domain-Driven Design on the origin of Behaviour Driven Development.
LO-2.5	K2	Summarize the reasons for working according to the principles of Behaviour Driven Development.
LO-2.6	K1	Remember the different work forms that can be used within BDD.
LO-2-7	K1	Recall the importance of Shared Understanding and Living Documentation within BDD.
LO-2.8	K1	Define what roles are involved for BDD.
LO-2.9	K2	Explain the role of the Three Amigos within BDD.
LO-2.10	K2	Summarize the process steps for Behaviour Driven Development.
LO-2.11	K2	Summarize which techniques are used in the process steps for Behaviour Driven Development.
LO-2.12	K1	Recall the end products or deliverables of the process for Behaviour Driven Development.

2.1 What is Behaviour Driven Development (BDD)

LO-2.1	K2	Summarize what Behaviour Driven Development (BDD) is.

Behaviour Driven Development was introduced by Dan North in 2006 (North, Introducing BDD, 2006). He describes it as a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters. (North, Agile Specifications, 2009)

Behaviour Driven Development (BDD) is an evolution of the ideas behind agile software delivery. With its roots in Test Driven Development, Domain-Driven Design, and Automated Acceptance Testing, BDD focuses on the ways an application is expected to work - its behavior.

Second-generation: BDD evolved from the principles of *Test Driven Development* (TDD), *Acceptance Test Driven Development* (ATDD), *Domain Driven Design* (DDD) and has elements of *Extreme Programming* (XP);

Outside-in: Instead of looking from the software application and how it acts or will act with the outside world, BDD is about understanding system usage from the outside in, focusing on end-users' behavior and how this behavior impacts the application (to be developed). An end-user can be either a person, another system or part of the code, a user interface, etc. (North, What's in a Story, 2022);

Pull-based: Only produce what is ready to be used by either an end-user, a system or code closer to a boundary or UI, following the Lean Software Architecture principle of 'just in time, just enough';

Multiple-stakeholder: Considering all stakeholders, both core stakeholders who define the vision (and often provide budget) and incidental stakeholders who support the solution delivery and may influence that solution; and last but not least the 'actors', who may be end-users or other systems interacting with the system.

High-automation: automation of acceptance- and unit tests whenever possible, to create a fast-feedback loop.

2.2 History of BDD

LO-2.2	K1	Recall the influence of Test Driven Development on the origin of Behaviour Driven Development.
LO-2.3	K1	Recall the influence of Acceptance Test Driven Development on the origin of Behaviour Driven Development.
LO-2.4	K1	Recall the influence of Domain-Driven Design on the origin of Behaviour Driven Development.

To understand Behaviour Driven Development, it is useful to know about its history. Behaviour Driven Development was a response to agile practices where comprehensive documentation became unimportant, as opposed to being just less valued over working software. It is called a second-generation methodology, following up on principles from e.g., Test Driven Development (TDD), Acceptance Test Driven Development (ATDD), Domain Driven Design (DDD) and Extreme Programming (XP).

Test Driven Development (TDD) is all about unit testing first. Before a programmer starts developing software, (s)he first starts writing unit tests. Only after finishing the unit tests the software will be developed until all unit tests succeed. Some refactoring may later be carried out to further improve the software, but no other functionality will be added.

The principles of TDD were expanded to the functional tests with Acceptance Test Driven Development (ATDD) which is often (mis)used as a synonym for BDD. With ATDD the functional acceptance tests are written first, then these are used to create the unit tests, and only after that step, the development of the new software starts until all tests succeed (both functional and unit

tests). Again, there is room for refactoring after all tests have succeeded. More on TDD and ATDD in : (Acceptance) Test Driven Development (introduction to Test Automation using BDD)Chapter 8.

Domain-Driven Design is a concept where the primary focus is on the business domain and its logic (Evans, 2003). A model of the domain serves as a basis for the software to be designed and therefore a creative collaboration between technical and domain experts is key.

2.3 Why is BDD useful?

LO-2.5	K2	Summarize the reasons for working according to the principles of BDD.
LO-2.6	K1	Remember the different work forms that can be used within BDD.
LO-2-7	K1	Recall the importance of Shared Understanding and Living Documentation within BDD.

Behaviour Driven Development (BDD) is a response to agile ways of workings failing to find a way to provide sufficient documentation. The *Agile Manifesto* claims that it values working software over comprehensive documentation (Agile Manifesto, 2001). In practice, this has often led to a misconception that documentation is not needed in agile environments. However, 'no comprehensive documentation' does not mean that 'no documentation' is needed at all.

Consequently, in many agile environments the test scripts are the first formalized documentation describing the functionality of the software. This can pose a huge risk, because the tests both describe what the software should be doing (according to its writer) as well as perform a check on that same description.

BDD has used this principle to elevate both documentation and test(automation) to a higher level. The documentation or the requirements are the test, hence the relationship with TDD. To mitigate the risks of tests being the only documentation (or vice versa), a set of rules have been created on how to get to these requirements and how to write them down. When this rule set is considered and the right people remain involved, BDD can ensure there is a single source of truth describing the functionality required and testing this functionality as well.

By using different working methods/techniques/strategies, such as *Impact Mapping*, *Event Storming*, *Example Mapping* and *Specification by Example*, the means are provided to make sure there is just enough documentation available, just in time. On top of that, the documentation is unambiguous and can be used to create a *shared understanding* between all stakeholders and other people involved. Just as for software development, the maintenance of documentation also becomes an iterative process, which makes it *Living Documentation*.

2.4 Who is involved in BDD?

LO-2.8	K1	Define which roles are involved for BDD.
LO-2.9	K2	Explain the role of the Three Amigos within BDD.

One of the most important aspects of Behaviour Driven Development is that it requires all stakeholders to be involved. Business, developers and possibly also operations (BizDevOps): all play a role. Creating a shared understanding is only possible if it is a joint effort. It closes the gap between business and IT, which will be explained in the following sections.

It is, however, not possible to have everyone involved all the time. One of the key concepts of Behaviour Driven Development is therefore the *Three Amigos* principle. The Three Amigos were originally one representative from the business, one representative from the software engineers and one representative from the test engineers. It has evolved into gathering the representatives of every stakeholder group involved, especially when discovering the first (often still high level) requirements. More about the Three Amigos in the next paragraph.

2.5 How does BDD work?

LO-2.10	K2	Summarize the process steps for Behaviour Driven Development.
LO-2.11	K2	Summarize which techniques are used in the process steps for Behaviour Driven Development.
LO-2.12	K1	Recall the end products or deliverables of the process for Behaviour Driven Development.

Behaviour Driven Development focuses on the end user. Therefore, BDD always starts with what an end user or a representative of this end user wants. This person could be a product owner, a subject matter expert or an actual end user of the functionality that is to be developed. In Figure 1, we simply refer to this person as 'Business'.

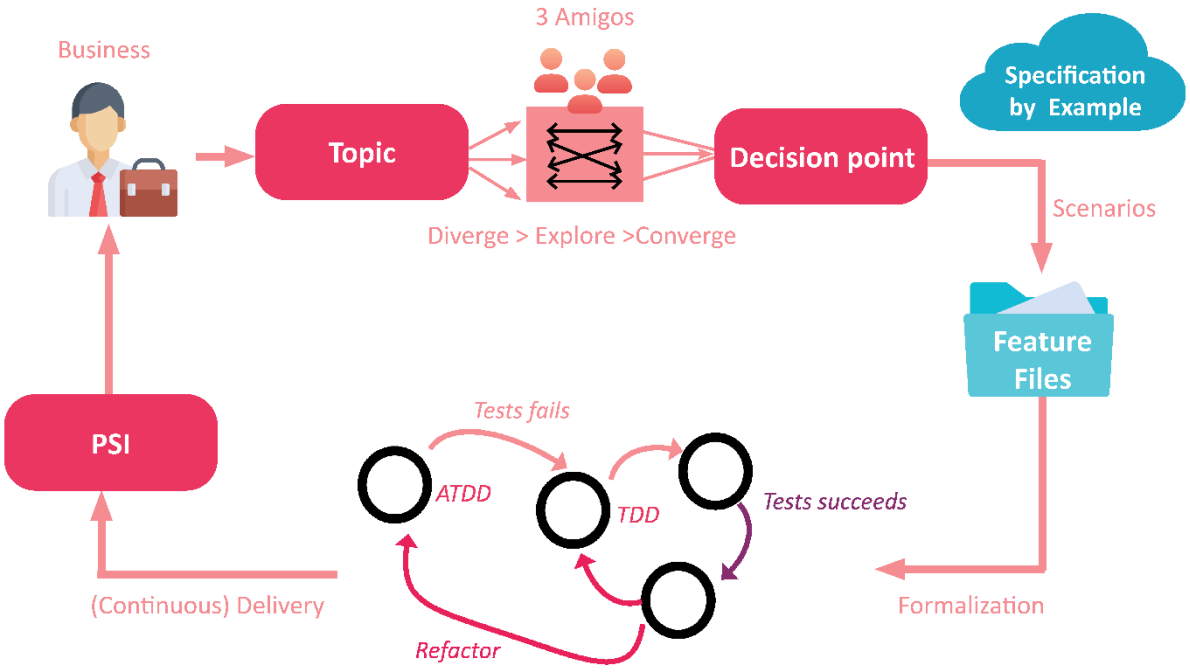


Figure 1: BDD overview

The business can come up with something they want. It could be in the form of a user story, a requirement, a wish, or demand. We call this a topic.

Before this topic is fully detailed and specified, it is discussed with representatives of the people involved with the development of this new functionality. One representative of each expertise is present; we call this discussion with representatives the Three Amigos (meeting).

A known method of discussing new functionality is by using the 'diverge-explore-converge' principle (see Figure 1: BDD overview).

The diverge-explore-converge principle means the topic is first discussed without any constraints, limitations, or boundaries, thus making the topic and its context bigger and bigger. Before it becomes too big, this discussion is stopped, and the topics that arose from it are to be explored, making connections between different ideas, and trying to find the logic behind the new functionality.

Finally, the generated broad ideas are narrowed down, or converged, to get to a decision point where the new functionality is small enough to be realized in a short time span (preferably one or at most a few iterations/sprints).

There are several tools and working methods that can be used to make the work method depicted in the diagram effective, e.g., Impact Mapping and Event Storming. These will be discussed in this syllabus later.

What is decided is then written down in scenarios using Specification by Example (see Chapter 5, Chapter 6 and Chapter 7). The scenarios are put into feature files and then formalized. This formalization makes it possible to use these feature files as input for Acceptance Test Driven Development and Test Driven Development.

The feature files now contain the actual requirements and are used to create (automated) functional acceptance tests, which are then used to create unit tests. At this point, the development team can start developing the new functionality which will eventually be delivered as a *Potentially Shippable Increment* (PSI) back to the business, where the 'increment' is a part of, or a whole product.

Chapter 3 : Impact Mapping

Impact Mapping is about asking the right questions in the right order to generate a shared understanding about the scope of a project or new functionality and to get to understand the purpose and goal of this new functionality.

Keywords

Impact Map, Impact Mapping, Why-Who-How-What-questions, Goal, Purpose, Impact, Actors, Deliverable, Problem space, Solution space

LO-3.1	K2	Summarize what Impact Mapping is.
LO-3.2	K1	Recall what Impact Mapping helps with.
LO-3.3	K2	Summarize the benefits of Impact Mapping.
LO-3.4	K2	Define the four building blocks used within Impact Mapping and classify what blocks belong to which problem space or zone of influence.
LO-3.5	K1	Remember what the 'Goal' block entails.
LO-3.6	K1	Remember what the 'Actor' block entails.
LO-3.7	K1	Remember what the 'Impact' block entails.
LO-3.8	K1	Remember what the 'Deliverable' block entails.
LO-3.9	K2	Identify the right participants for Impact Mapping.
LO-3.10	K2	Interpret what is needed to perform Impact Mapping.
LO-3.11	K2	Summarize how Impact Mapping works.
LO-3.12	K1	Recall each step/question for Impact Mapping.
LO-3.13	K2	Summarize the typical pitfalls when performing Impact Mapping.
LO-3.14	K3	Apply Impact Mapping in a simple situation.

HO-3.1	HO-2	Using Impact Mapping in a team using an example case.

3.1 What is Impact Mapping?

LO-3.1	K2	Summarize what Impact Mapping is.
LO-3.2	K1	Recall what Impact Mapping helps with.

An *Impact Map* is a visualization of scope and underlying assumptions, created collaboratively by (senior) technical and business people. It is a mind map grown during a discussion facilitated by answering the questions *Why? Who? How?* and *What?* (in this order).

Impact Mapping is based on user interaction design (how will the user interact with the (future)

product), outcome driven plan making (planning for prioritized product or features) and mind mapping. It can be used as a visual strategic planning method to decide which features to build into a product. As it begins with the intended *goal* (Goldratt, 1984) and from there, identified features will have a direct and concrete impact on achieving that desired goal and a clear rationale why.

Impact maps help delivery teams and stakeholders visualize roadmaps, explain how deliverables connect to user needs, and communicate how user outcomes relate to higher level organizational goals (Adzic, Impact Mapping Website, 2021).

Impact Mapping was developed by Gojko Adzic (Adzic, Impact Mapping, 2012).

3.2 Why is Impact Mapping useful?

LO-3.3	K2	Summarize the benefits of Impact Mapping.
LO-3.4	K2	Define the four building blocks used within Impact Mapping and classify what blocks belong to which problem space or zone of influence.
LO-3.5	K1	Remember what the 'Goal' block entails.
LO-3.6	K1	Remember what the 'Actor' block entails.
LO-3.7	K1	Remember what the 'Impact' block entails.
LO-3.8	K1	Remember what the 'Deliverable' block entails.

Systems are not developed just for the sake of developing them. A system is developed with a *purpose* in mind, or to reach a goal, which is to support one or more business processes (and not the other way around). However, development teams may not be close enough to the business to fully understand the actual reason why the organization needs this function or system. This may have a (negative) effect on the planning, setting priorities, as well as the delivered functionality and other quality characteristics. Furthermore, development teams are often result-oriented, overlooking the proper reasoning behind the 'system' or change.

Impact Mapping is based on the four building blocks shown below in Figure 2.

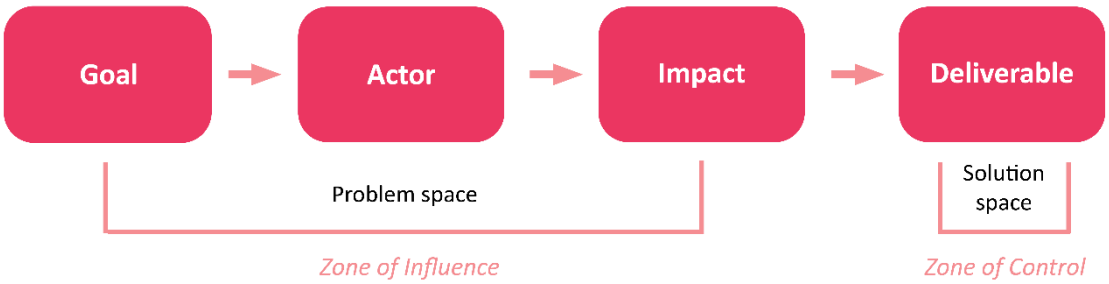


Figure 2: Building blocks of Impact Mapping

The starting point is the 'Goal': what is it you want to achieve? In other words, the WHY. For instance: "We want to increase sales". (More on the WHY question in section 3.4). In some contexts, this means that you have to take one step back, look at the bigger picture and then work your way forward again by applying the following steps.

In most cases, the goal cannot be reached in just one single way. There will be multiple changes that will impact the increase of sales. But before diving into the 'Impact' you'll need to define the '*Actors*' whose behaviors will be impacted or who can produce or obstruct the desired effect. The actors can be identified by asking WHO. This is not limited to the actual or 'primary' users of the system to be developed, but it may also involve indirect actors. (More on the WHO question in section 3.4).

Once you have established who the actors are who can (significantly) influence the success of a project or product milestone, it is time to focus on the impact by answering the HOW question. (More on the HOW question is section 3.4).

The goals, actors and impacts are also called the 'Problem Space' or the zone of (what we want to) influence.

Once you know what the problem is and what and how you want to influence, only then you can start thinking about the Solution Space: where you actually are in (the Zone of) Control by creating the 'Deliverables'. These deliverables can be identified by asking the WHAT question. It will result in e.g., a new or updated system that will help making the impact by changed behavior, working towards the goal.

The result of Impact Mapping will show that there are indeed several roads (deliverables) that lead to Rome and also that not all of these roads may be as effective. Now that Impact Mapping has identified multiple roads, we can gather feedback and give priority and direct our effort to the Actors impacted most and Deliverables that will have the greatest Impact. Similar to the map metaphor that Gojko uses: if there is an obstruction, choose a better alternative (Adzic, Impact Mapping, 2012).

Impact Mapping prevents organizations from losing their focus or from focusing too much on the deliverables while building products and delivering projects and forgetting the impact to be made, who their projects should impact and, most importantly: what goal to reach.

In addition, by clearly communicating things that might yet be assumptions, teams will be able understand and adjust or remove assumptions, and better align their activities with actual business objectives and as such make better roadmap decisions.

3.3 What is needed for Impact Mapping and who is involved?

LO-3.9	K2	Identify the right participants for Impact Mapping.
LO-3.10	K2	Interpret what is needed to perform Impact Mapping.

Impact Mapping is not just a single session or (planning) workshop; rather, it is a mindset (and/or a way of working) that should be implemented into a project. However, the four base questions 'Why?', 'Who?', 'How?' and 'What?' can be integrated into the Three Amigos meetings, refinement sessions and/or other planning sessions. It is recommended to start with Impact Mapping before having any other discussion on the requirements. In that sense, it is highly recommended to make the first Impact Map during the Three Amigos session (or even prior to that).

Impact Mapping is a way to facilitate a discussion, there are no tools required, although we can recommend some basics needed to create a successful Impact Map:

- Suitable room, quiet and large enough to contain the modelling surface. It should contain a (large) wall;
- Writable surface, most likely a white board or brown paper roll, or its digital version;

- Some sticky notes. There is no preferred coloring, but it may be useful to choose different colors for the Goals, Actors, Impact and Deliverables;
- Some (Whiteboard) markers, ideally one per participant plus backup;
- The right people: (representatives of) the relevant stakeholders from both the development and business areas (Three Amigos), or all the stakeholders. For the sake of having an effective discussion, the group of people should not be too big, which means a maximum of 6 to 8 people;
- A facilitator.

3.4 How does Impact Mapping work?

LO-3.11	K2	Summarize how Impact Mapping works.
LO-3.12	K1	Recall each step/question for Impact Mapping.
LO-3.13	K2	Summarize the typical pitfalls when performing Impact Mapping.
LO-3.14	K3	Apply Impact Mapping in a simple situation.

HO-3.1	HO-2	Using Impact Mapping in a team using an example case.

To organize an effective Impact Mapping session, the following steps shall be executed:

- **Invitation:**
  Invite participants to the workshop. Select the right people: as stated in 3.3: there should be a good mix of participants to answer the WHY, WHO, HOW and WHAT questions.
- **Organize room and material:**
  When organizing a physical session, a room with a whiteboard or wall for the brown paper is required. If during the session the model becomes bigger than the board, adjust the modelling space with whatever is available to eliminate the space limitation or split the goal and zoom in on the topics one by one. It is recommended to organize the sessions as a time box to prevent potential endless debates.
- **Explore and answer the WHY, WHO, HOW and WHAT questions**
  The actual session of Impact Mapping is centered around 4 questions, in this order:

  WHY? - Why are we doing this? What is the goal we are trying to achieve?

  WHO? - Whose behavior do we want to impact? Who can produce the desired effect? Who can obstruct it? Who are the consumers of our product? Who will be impacted by it? These can be divided into primary actors (whose goals are fulfilled), secondary actors (who provide services) and off-stage actors (who have an interest in the behaviors but are not directly benefiting or providing a service)

  HOW? - How should our actors' behavior change? How can they help us to achieve the goal? How can they obstruct or prevent us from succeeding?

  WHAT? – What can we do, as an organization or a delivery team, to support the required impacts?

In a group discussion, by answering these four questions, the **goal**, the **actors**, the **impacts,** and the **deliverables are successively determined**. For the facilitator, it is important to follow this order, as depicted in Figure 3: Diagram of Impact Mapping.

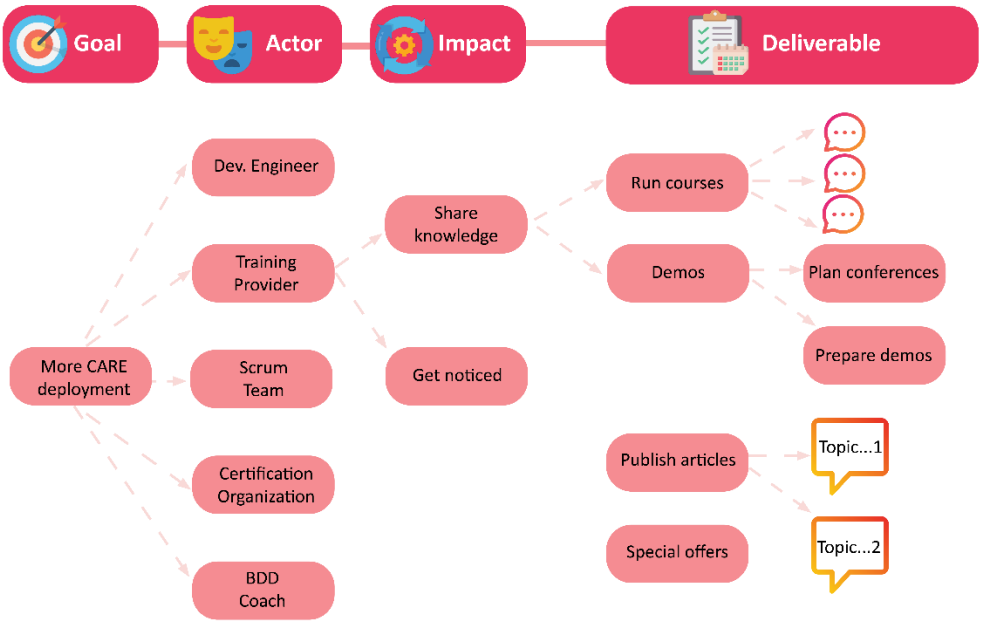


Figure 3: Diagram of Impact Mapping

Additional remarks

Note that Impact Mapping is rather simple to explain, but it is not easy to perform. The main reason for this complexity is that there are some common pitfalls one should avoid:

- Deliverables are not only in the zone of control but also in the comfort zone of most people. It is tempting to immediately jump to deliverables (implementation), but without knowing what the actual goal is, what (behavioral) change we want to achieve, and what will be the (expected) impact. Without this insight, determining the deliverables is less valuable.
- Identifying stakeholders can be difficult. Primary actors will usually be rather easy to identify but the secondary and possible off-stage (indirect) actors are often more difficult to specify. A supporting method like Value Stream Mapping may be of help. With this method sources (of value and change) and 'tributers' (both contributors and polluters) will be identified.
- Identifying impacts can also be difficult. Impact Cards may be of help by determining per 'Actor', what he/she 'Will Do' differently 'Instead of' their current behavior.

As a result of the collaborative creation of the Impact Map, and thus of overcoming these pitfalls, there will be no more debate regarding the Why, Who, How and What answers later on, except in the case new insights would arise.

Chapter 4 : Event Storming

Event Storming is a workshop that can be used to translate the Impact Map into actual events. This may result in a workflow showing how the functionality that is to be developed is going to work and making its impact on the involved actors more concrete.

Keywords

Event Storming, Business process modelling, Domain events, User, Command, System, Policy, Event, View Model

LO-4.1	K2	Summarize what Event Storming is.
LO-4.2	K1	Remember the link between Event Storming and Domain-Driven Design.
LO-4.3	K2	Summarize the benefits of using Event Storming.
LO-4.4	K2	Classify the different scenarios in which Event Storming can be applied.
LO-4.5	K2	Identify the right (amount of) participants for Event Storming.
LO-4.6	K2	Interpret what is needed to perform Event Storming.
LO-4.7	K2	Summarize the process steps to organize an Event Storming session.
LO-4.8	K1	Remember the process diagram of Event Storming.
LO-4.9	K1	Remember the additional steps for Event Storming.
LO-4.10	K1	Recall the points of attention for an Event Storming session.
LO-4.11	K3	Apply Event Storming in a simple situation.

HO-4.1	HO-2	Using Event Storming in a team using an example case.

4.1 What is Event Storming?

LO-4.1	K2	Summarize what Event Storming is.
LO-4.2	K1	Remember the link between Event Storming and Domain-Driven Design.

Event Storming is a flexible workshop format for the collaborative exploration of complex business domains that helps find out what is happening in the domain of a software program. It allows to identify and visualize issues which have not yet been resolved.

It is a lightweight workshop, which intentionally requires no support by a computer. It is designed to enhance creativity and open up communication. The business process is “stormed out” as a series of domain events. The result is that the business process is expressed in the form of sticky notes on a wide wall.

The basic idea is to bring together software developers and domain experts so that they can learn from each other. To make this learning process easier, event storming is meant to be fun. Its name

was chosen to indicate that its focus should be on the domain events, and that the method works similarly to brainstorming.

The method was formalized in 2013 by Alberto Brandolini in the context of Domain-Driven Design (DDD) (Brandolini, 2013).

4.2 Why is Event Storming useful?

LO-4.3	K2	Summarize the benefits of using Event Storming.
LO-4.4	K2	Classify the different scenarios in which Event Storming can be applied.

The goal of the Event Storming workshop is to learn as much as possible in the shortest possible time and to visualize the (business) process once the impact of a new product/project is already known (using for instance Impact Mapping, see Chapter 3). Event Storming is a means for *business process modelling* and requirements engineering, but can also be used for retrospectives, root cause analysis or the induction of new workers/employees.

The adaptive nature of Event Storming allows for strong cross-discipline conversations between stakeholders with different backgrounds and helps create a new type of collaboration beyond discipline and specialization boundaries.

Both the people with the questions and the people who know the answers (domain experts and software developers) are brought together to inform and learn from each other by engaging people from and into the whole process. In an open manner, they can build the process model and get insight and overview.

Event Storming is powerful: it allows practitioners to come up with a comprehensive model of a complete business flow in hours instead of weeks, and without wasting their time. It is efficient, as the resulting model allows for a quick determination of Context and Aggregate boundaries.

Furthermore, the basics of Event Storming are easy. The notation is ultra-simple. No complex jargon or modelling languages are used that might cut off participants from the heart of the discussion. It is fun to lead, but also to participate in the workshops, as people are energized and deliver more than they expected. The right questions arise, in the right atmosphere, enabling ‘cross-perspective conversations’.

Event Storming can come in different flavors to be used in different scenarios:

- To assess the health of an existing business process and to discover the most effective areas for improvements;
- To explore the viability of a new start-up business model;
- To envision new services, that maximize positive outcomes to every party involved;
- To design clean and maintainable Event-Driven software, to support rapidly evolving businesses.

4.3 Who is involved in Event Storming?

LO-4.5	K2	Identify the right (amount of) participants for Event Storming.

Event Storming does not require specific roles, as it is highly dependent on the context. A good mixture of curiosity and wisdom should be involved:

- participants who know which questions to ask (and who are curious about their answers);
- participants who know the answers.

In most cases, there is a need to develop an (automated) solution for a problem. In such cases, 'business', analysts, or user experience (UX), service etc., can explain what the problem is.

Developers and testers will most likely be the ones asking questions regarding what the actual problem is, and whether there are any rules or policies that need to be implemented. This implies basic paths but also corner cases. Business (representatives) should be able to answer these questions.

To make this rather unconventional type of meeting successful, it may require some effort to get people 'on board', but the more questions are answered at an early stage, the less time and money is wasted at a later stage. Usually, once an Event Storm / Event Storming workshop starts to get shape, participants get enthusiastic about it and will see the true goal and purpose of the event storm.

In an Event Storming workshop ideally 6 to 8 people participate, but this number could easily be raised to approximately 30 people; which is quite unique and unheard of in the context of many other working methods for workshops.

4.4 What is needed for Event Storming?

LO-4.6	K2	Interpret what is needed to perform Event Storming.

The exact requirement for a successful Event Storming meeting is dependent on the maturity and experience of the organization with Event Storming.

Some basic requirements are:

- Suitable room, quiet and large enough to contain the modelling surface, think about a (large) wall;
- Writable surface, most likely a white board or brown paper roll, or its digital version
- Sticky notes, in different colors and preferably:
  - Rectangular sticky notes in the colors (pale) yellow and lilac,
  - Square sticky notes in the colors: orange, blue, yellow, red and green;[2]
- (Whiteboard) markers, ideally one per participant plus backup;
- The right people: see 4.3;
- Facilitator.

4.5 How does Event Storming work?

LO-4.7	K2	Summarize the process steps to organize an Event Storming session.
LO-4.8	K1	Remember the process diagram of Event Storming.
LO-4.9	K1	Remember the additional steps for Event Storming.
LO-4.10	K1	Recall the points of attention for an Event Storming session.

[2] The shapes and colors don't have any specific meaning, so you can just use any other color sticky note as well. These are however the standardized colors as prescribed by Brandolini when introducing Event Storming (Brandolini, 2013). Advantage of using these is that there is no need to explain the meaning of each color to a possible newcomer.

LO-4.11	K3	Apply Event Storming in a simple situation.

HO-4.1	HO-2	Using Event Storming in a team using an example case.

To organize an effective Event Storming session, the following steps shall be executed:

- **Invitation:**
  Invite the participants to the workshop. Select the right people: as stated in 4.3, there should be a good mix of participants who know the questions that should be asked and who know the answers.
- **Organize room and material:**
  Unlimited modelling space is required. Complex problems may not be properly analyzed because there is not enough space available to draw the whole problem. So, a room shall be reserved that contains large (empty) walls and /or that offers the possibility to stick large sheets of (white or brown) paper on the wall. If, during the session, the problem requires more space than the available board allows for, then adjust the modelling space with whatever is available to eliminate the space limitation. It is recommended to organize the sessions as a time box.
- **Explore the domain, starting with *Domain Events*:**
  A Domain Event is something meaningful happening in the domain. It is recommended to not directly translate this event into software, to make it more understandable for non-technical people.
  An event may be the predecessor or successor of another event. All events shall be written on sticky notes (according to the convention, orange sticky notes, and the past tense shall be used) and placed onto the modelling surface, according to a 'timeline'. Timelines from different departments will become visible, as well as interrelations and/or dependencies.
  The intention here is not to focus on a specific area, as this will not allow us to uncover the big picture.
  There is not a single format for the workshop. The first steps are (more or less) the same, but the format may vary according to the roles of the participants, and to the project scope.
- **Explore the origin of Domain Events:**
  Events may be the direct consequence of a user action. Such a *User* is represented by a square yellow sticky note. If more types of users exist, 'personas' may be used (Nb. a Persona is a name or other identification for a specific (type of) end user of a system).
  The actual *Command* is represented by a blue sticky note as 'input' to the event. Such a command can also be initiated by another (external) system or e.g., time based.
  Where the command is given to a *system* (being a machine, software or a human executing an action), this is represented by a rectangular yellow sticky note.
  In some cases, events will be the direct consequence of other events. These events are then located closely together.
  Where decisions are made for or by a process, there will be *policies* that drive that decision. Such policies are represented by lilac sticky notes.
  Outcomes of the event and/or policy are presented to a user or other system in what is called a *View Model*, represented by a green sticky note.
  In case there are issues or questions that cannot be solved or answered directly, they are also written down on red sticky notes (alternative: neon pink). 'Red areas' indicate that these topics need further clarification, possibly by additional participants. These conflict areas are also referred to as 'hotspots'. The sticky notes always follow this strict order as is also presented by

the diagram shown in Figure 4.

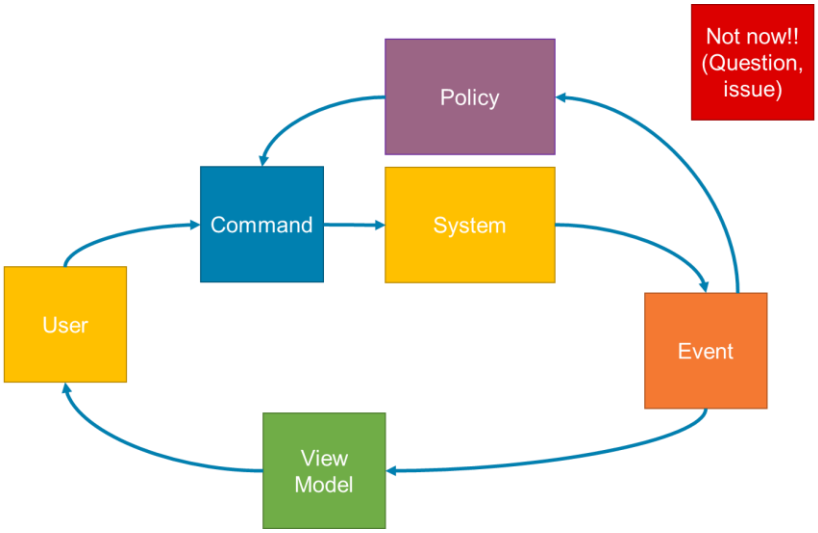


Figure 4: Diagram for Event Storming

- **The big picture:**
  Due to the 'unlimited' space, the 'big picture' becomes visible, as it comprehends the whole 'structure' of people, systems, processes, narratives, stories (more on this in Chapter 6), user journeys, value generation paths: what happens in the flow. It all appears in a simple and understandable notation that will look something like what is shown in Figure 5.

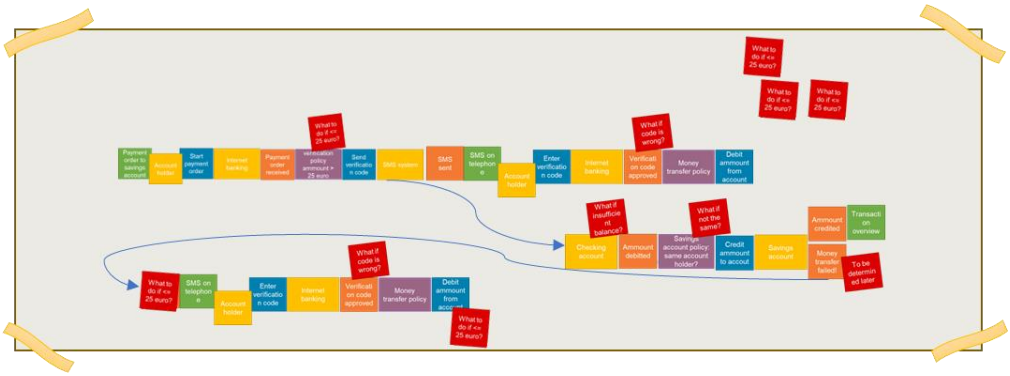


Figure 5: The big picture of an Impact Map

The process of creating the picture will initiate conversations to support further insights. E.g., by using dot voting (where each participant has a fixed number of dots to place on certain sticky notes), the most important bottlenecks can additionally be identified if consensus is not yet reached. Keep in mind that Event Storming is independent of what should be done to solve the problem (backlogs, estimates etc.).

- **Look for Aggregates:**
  Aggregates group various commands, events, and reactions together, in a logical manner. It helps craft software in a way that fully respects relationships within business processes. Instead of defining aggregates starting from code, take an outside-in approach: the *Aggregate* is the portion of the system that receives commands and decides whether to execute them or not, thus producing a *domain event*. Look for state machine logic or behavior, do not look for data and try to postpone specific naming. Aggregates are represented by rectangular pale yellow sticky notes.

Optional additional steps

In addition to the basic steps of the original Event Storming, additional goals or hotspots may become clear. As stated, any key issues in the flow (questions, discussions, contradictions, etc.) shall be written down on *red sticky notes* near the area where they arise. These new 'targets' are worth considering as a rewarding detour from the standard route:

- **Exploring Subdomains**: allowing domain experts to show more expertise in an area that was not expected, but also leaving other portions of the domain to others. Areas of responsibility normally map well to different subdomains.
- **Exploring Bounded Contexts**: Within the process flow there may appear some 'pivotal events': the most significant events in the flow. These are a trigger to draw boundaries (vertical lines where the pivotal event is) and investigate interfaces between the multiple consistent models that will coexist in domains. To be flexible, the borders can be drawn using masking tape.
- **Sketching Key Acceptance Tests**: When talking about corner cases or ambiguous scenarios, ambiguity can be removed by defining acceptance tests or asking and answering questions like "What would happen if…? ". Not for all scenarios, but only for tiebreakers, it is a way to capture the emerging knowledge right away.
- **Sketching Key Read Model Artifacts**: for those scenarios where what the user will see is more important than what the system does. Screens, tables, or graphs which are particularly valuable to a given user to perform an action or make a decision, can be sketched on a green sticky note, and placed close to the command it is associated to.

Points of attention:

Structure: Do not expect to organize a structured workshop. When you have a hint, tip, or question, write it on a sticky note and put it near the hotspot.

Language: Do not focus on consistent language from the beginning. Different disciplines are collaborating using different terminologies. Consistency should be introduced slowly because it is difficult to work towards an 'ubiquitous language'.

Completeness: Do not strive for model completeness. The model is going to be big. Completing it will add only little more value and will cost too much time, it may even be scary for some participants. Accepting some degree of incompleteness will make the workshop less boring and more fruitful, also keeping the 'Pareto principle' in mind: 80% of consequences come from 20% of causes (Juran & Blanton Godfrey, 1999).

Disciplines: When exploring for software development only, the result may be even more powerful. Aggregates, Commands and Domain Events, all map well into software artifacts. The big picture may become clear quickly and more focus may be put on the flow within the system. However, a focus on software development only may lead to losing the connection with the business and business value!

Chapter 5 : Specification by Example - The Basics

This chapter describes the basics of Specification by Example, how it works, who are involved and what we need to implement it. It will be further elaborated in the next chapters which will describe Example Mapping, Specification with Examples and (Acceptance) Test Driven Development.

Keywords
Specification by Example, Behaviour Driven Development, Shared understanding, Single source of truth, Living documentation, Feature file, Examples, Scenarios, Automate validation, Validate frequently, Executable specification

LO-5.1	K2	Summarize what Specification by Example is.
LO-5.2	K1	Remember what the goal and purpose of Specification by Example is.
LO-5.3	K2	Summarize the benefits of Specification by Example.
LO-5.4	K2	Identify the right participants for Specification by Example.
LO-5.5	K2	Interpret what is needed to perform Specification by Example.
LO-5.6	K2	Summarize the process steps for Specification by Example.
LO-5.7	K2	Interpret the process diagram of Specification by Example.
LO-5.8	K3	Apply Specification by Example in a simple situation.

5.1 What is Specification by Example?

LO-5.1	K2	Summarize what Specification by Example is.
LO-5.2	K1	Remember what the goal and purpose of Specification by Example is.

Specification by Example was originally described by Gojko Adzic (Adzic, Specification by Example, 2012). Specification by Example (SbE) is a collaborative approach used to define requirements and business-oriented functional tests for software products, based on capturing and illustrating requirements, using realistic examples instead of abstract statements

. It is applied in the context of agile software development methods, in particular *Behaviour Driven Development*. This approach is particularly successful for managing requirements and functional tests on large-scale projects of significant domain and organizational complexity.

The ultimate goal of Specification by Example is not limited to building the software right, it is also to build the right software. It is all about creating *shared understanding* using a *single source of truth* which can eventually become the *Living Documentation* which will also be used for the (automated) validation of the software which is to be delivered.

5.2 Why is Specification by Example useful?

LO-5.3	K2	Summarize the benefits of Specification by Example.

The most important reasons to start using Specification by Example are:

- Shared Understanding
- Single Source of Truth
- Automating validation and validating frequently
- Evolving to living documentation

Specification by Example uses domain language and consists in writing the specification or requirements down using very specific, concrete examples. These examples should be readable for everyone in the project and form the basis for the software to be tested and developed. Because it is understandable for everyone in the project, both business and development, a shared understanding is created. Because it contains requirements, tests and eventually (living) documentation, a single source of truth is created. The *feature file* is the (plain text) file where the *examples* are written down in *scenarios* and is the only place where these requirements/tests are documented. The scenarios can also be used to create tests and, using tooling, they can be used to form the framework for test automation, which means specification by example can (also) be used to *automate validation* and therefore *validate frequently*.

5.3 Who is involved with Specification by Example?

LO-5.4	K2	Identify the right participants for Specification by Example.

As mentioned before, Specification by Example aims to create shared understanding for everyone involved in the development of the software, both business (representatives) and the development team, and possibly more actors or stakeholders. It is thus essential to have every stakeholder involved at some point during the process. Quite often, someone from the business is the primary responsible (product owner, business analyst, functional designer); later, more people are involved (for instance Three Amigos) and at a later stage, everyone from the development team and possible other stakeholders get involved (See paragraph 5.5 for more details on the involvement of actors in this process).

5.4 What is needed for Specification by Example?

LO-5.5	K2	Interpret what is needed to perform Specification by Example.

No special tools are required to perform Specification by Example. The scenarios will be written down in plain text, which can be done using any text editor or even just pen and paper. When discussing the scenarios with a group (the Three Amigos or more), it can be useful to have a large screen present where everyone can read along or use a flip-over/whiteboard with markers. When using Example Mapping creating the examples, some additional materials are needed. For more information on what is needed, see Chapter 6 and Chapter 7.

5.5 How does Specification by Example work?

LO-5.6	K2	Summarize the process steps for Specification by Example.
LO-5.7	K2	Interpret the process diagram of Specification by Example.
LO-5.8	K3	Apply Specification by Example in a simple situation.

Specification by Example is a way of working for which the goal is to work from a defined business goal and scope to refined Specifications with Examples, which can then be used for (test) automation. Using the examples for test and test automation turns them into *executable specification*, which is then the basis for your Living Documentation. This is visualized in Figure 6.

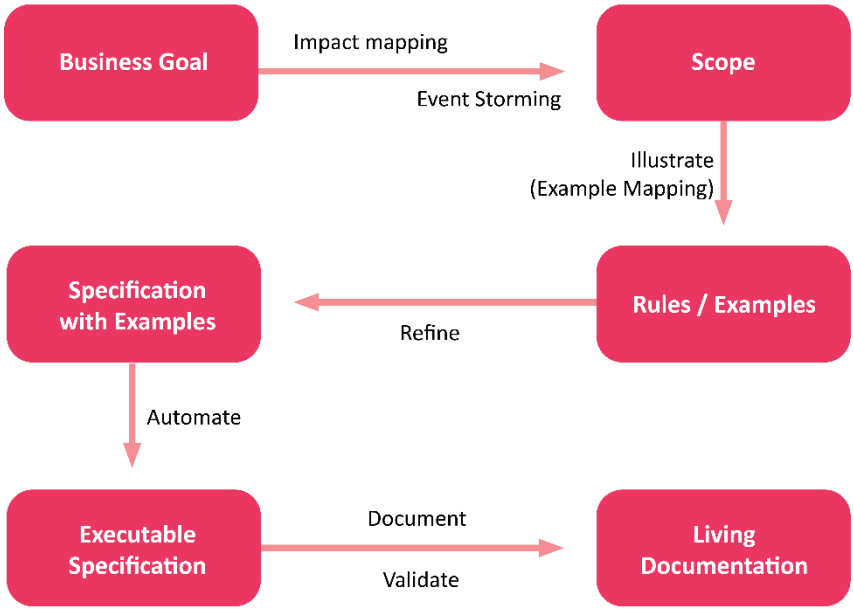


Figure 6: Specification by Example and its position within the CARE process

Specification by Example is a collaborative approach, which means it is all about working together. The end goal is to have specifications written down in the most detailed and concrete way as possible, to prevent any misunderstanding or misinterpretation from happening now or at a later stage. This means there is a lot of effort needed to write these specifications down properly. This step can be challenging when there are many people involved, who all have opinions on how exactly one should write them down. It is thus a good practice to start small and further elaborate on it and expand as time progresses. In CARE, Specification by Example is divided into a few different steps where the first step is to start with Example Mapping (see : Example Mapping). During Example Mapping a story is refined into a number of rules which are then further refined into scenario titles. This can be prepared by one person, for instance a business analyst, and the Example Mapping can be executed together with the Three Amigos or the entire development team and all the relevant stakeholders.

When the different scenarios are identified, the next step is Specification with Examples. During this second step, the actual Given-When-Then scenarios are written down (more on these scenarios in Chapter 7). These scenarios can be prepared by, for instance, the business analyst and/or the tester, only to be discussed again with either the Three Amigos, or the entire development team and all the relevant stakeholders. It is also possible to split this second step into two sessions, one with the Three Amigos, during which the scenarios are discussed and written down in a draft version, then a second meeting during which the Three Amigos present the scenarios to the rest of the team and stakeholders so the scenarios can be finalized into a final version.

Once the scenarios have been finalized, they can be used to start implementing the different tests and test automation (More on this step in Chapter 8)

: (Acceptance) Test Driven Development (introduction to Test Automation using BDD).

Chapter 6 : Example Mapping

While Impact Mapping and Event Storming consist of describing an entire process, Specification by Example goes into the concrete, small details. Example Mapping is a collaborative approach to converting the impact *and* events *into a* story*, converting the story into* rules *and describing these rules with* examples*. Example Mapping is essentially the link that ties all these processes together.*

Keywords

Examples, Example mapping, Story, Rule, Deliberate discovery, Acceptance criteria, Known unknowns, Unknown unknowns, Requirements, Context – Action – Expected outcome

LO-6.1	K1	Recall what Example Mapping is.
LO-6.2	K2	Summarize why Example Mapping is beneficial.
LO-6.3	K1	Select which roles are typically involved with Example Mapping.
LO-6.4	K1	Choose the right equipment needed for Example Mapping.
LO-6.5	K2	Summarize the process steps for Example Mapping.
LO-6.6	K1	Recall the outcomes of Example Mapping.
LO-6.7	K1	Remember the visualization of Example Mapping.
LO-6.8	K2	Compare the two approaches (Bottom-Up / Top-Down) for (deliberate) discovery in Example Mapping.
LO-6.9	K3	Apply Example Mapping in a simple situation.

HO-6.1	HO-2	Using Example Mapping in a team using an example case.

6.1 What is Example Mapping and why is it useful?

LO-6.1	K1	Recall what Example Mapping is.
LO-6.2	K2	Summarize why Example Mapping is beneficial.

Examples play an important role as they are of great value because they can be used from the specification stage, through development, to testing. Examples support the team's shared understanding about the context and desired changes. Examples help spotting inconsistencies and functional gaps. They should describe the end user's behavior – never the implementation. Examples should be realistic, precise, complete, and easy to understand (see also Chapter 2

: Specification by Example - The Basics).

To support defining and elaborating examples, Example Mapping is a selected method used in the context of BDD.

Examples are a powerful way to help explore the problem domain and to define and describe functionality. It forces you or the team to think things through, think about wanted or unwanted implications of design choices and to discover how functionality might (or might not!) actually work in real life: *deliberate discovery*. The goal of using examples is to create a shared understanding within the team of how the system (and its users) should behave, using a shared vocabulary, and to reach a common agreement on what to build (and test).

Example Mapping is a working method used to clarify a *story* with several (business) *rules* and to make these rules concrete by adding *examples* to the story. It is a time-boxed low-tech method aimed to help software teams gather details on what to build by identifying and drafting examples of how the system should behave. Through conversation, the *acceptance criteria* can be clarified and confirmed, which also means they form a great basis for acceptance tests.

Examples and Example Mapping make this conversation more efficient/effective, and they offer a powerful, productive means to discover what you do not know yet (deliberate discovery): the '*known unknowns*', and, maybe even more importantly, the '*unknown unknowns*', such as new business rules or refining existing business rules. From that perspective, Example Mapping, as a result, can be used to break up 'big' stories into smaller pieces of (future) functionality.

Example mapping aims to search for examples that are unique and that are chosen for a specific reason. It helps creating just enough examples: no more than absolutely needed and not too few so there might be gaps in the specified functionality. However, it can be difficult to come up with examples and to know when you have a sufficient number of examples (not too many, not too few).It can also be challenging to convert certain business rules into examples.

Note: the 'stories' mentioned here are not to be confused with 'user stories' which are used to manage work packages in an agile way of working. Example Mapping may however support or even replace the backlog refinement. When the stories are understood better, it can shorten sprint planning.

Example Mapping was formalized in 2015 by Matt Wynne in the context of Behaviour Driven Development (Wynne, 2015).

6.2 Who is involved in Example Mapping?

LO-6.3	K1	Select which roles are typically involved with Example Mapping.

Example Mapping does not require specific roles to be involved, but it should at least involve people who have a stake in the software to be developed. Once again, it depends on the context of the project.

Typical roles involved are Product Owner and/or business analyst, developer, tester, and optionally a customer, UX expert, service, support, etc. who have input or stake in how the software and its users should behave, and as such can come up with, compose or complete examples.

6.3 What is needed for Example Mapping?

LO-6.4	K1	Choose the right equipment needed for Example Mapping.

Some basic requirements are:

- Suitable room, quiet and large enough to contain the participants;
- A flat surface to lay out the index cards (optionally a writable surface);
- Colored index cards (alternatively rectangular sticky notes) can be used. Preferred colors: yellow, green, blue, and red;
- (Whiteboard) markers or 'sharpies', one per participants plus backup;
- A stopwatch or kitchen timer (for time boxing);
- The right people: see 6.2;
- A facilitator, especially when starting out with Example Mapping;
- Optionally: a story, or event storming result as subject or input for the workshop.

6.4 How does Example Mapping work?

LO-6.5	K2	Summarize the process steps for Example Mapping.
LO-6.6	K1	Recall the outcomes of Example Mapping.
LO-6.7	K1	Remember the visualization of Example Mapping.
LO-6.8	K2	Compare the two approaches (Bottom-Up / Top-Down) for (deliberate) discovery in Example Mapping.
LO-6.9	K3	Apply Example Mapping for a simple situation.

HO-6.1	HO-2	Using Example Mapping in a team using an example case.

Example Mapping can be run during Three Amigos sessions, deliberate discovery sessions, separate sessions or during refinement. Deliberate discovery sessions can be introduced for complex functionality where the regular three amigos and refinement sessions aren't enough to discover all (possible) implications of the new functionality. They are solely about discovering known unknowns and unknown unknowns. With Example Mapping the more people (stakeholders) join, the more information is shared. However, to maintain a good pace and avoid endless debates, it is recommended not to have too many people involved.

Per described story (derived from the Impact Map and Event Storm) one or more examples are drafted and discussed, grouped by (business) rules. Depending on situation and context you can start with the rules (top-down discovery) or start with the examples (bottom-up discovery), see also Figure 7). During Impact Mapping besides the (obvious) examples we describe:

- Rules that summarize a group of examples, (business) rules, or other agreed constraints about the scope of the story;
- Questions about scenarios which answer or outcome nobody in the conversations knows; or assumptions made in order to progress;
- New (user) stories discovered or sliced and deferred to out of scope.

The process consists of the following steps and shouldn't take more than circa 25 minutes per story:

1. Start with writing the story under discussion on a yellow card and placing it at the top of the table (or another surface). Alternatively, start with drawing up rules and decide how to logically group these into one or more (new) stories.

2. Write each of the acceptance criteria, or (business) *requirements* or rules that are already known, on blue cards and place them across the surface, beneath the yellow story card.
3. For each rule, write one or more examples to illustrate it. Write these examples on green cards and place them under the relevant rules.
   The recommended syntax for the examples is *Context – Action – Expected outcome* (for instance: '(The one where) a new CARE course is available, and students sign up for it successfully'). Drawings are also allowed. There is no need for systems or user interfaces. Think about positive (good weather) but also negative (bad weather) examples.
4. Discuss these examples, uncover questions that nobody in the room can answer. Questions are captured on red cards, so they can be answered later, and the conversation can continue without interruptions.
5. Continue until the group agrees that the scope of the story is clear, or when the time box of around 25 minutes expires (Wynne, 2015). At that stage, each role draws their conclusions: the business representative considers whether there is anything missing; the developer considers whether they have the information needed to start working on it, and the tester considers whether they can test the system at that point.
6. The group decides whether they need to continue with this step, or the story is ready to pull into development, e.g., by using thumb voting. If there's no agreement on this yet, an additional session can be organized.
7. It is recommended to save the results, e.g., by taking pictures of the results.

Note that following the above order is not mandatory or fixed. Example Mapping is flexible like all the working methods within CARE. The goal of the process is leading, not the process itself.

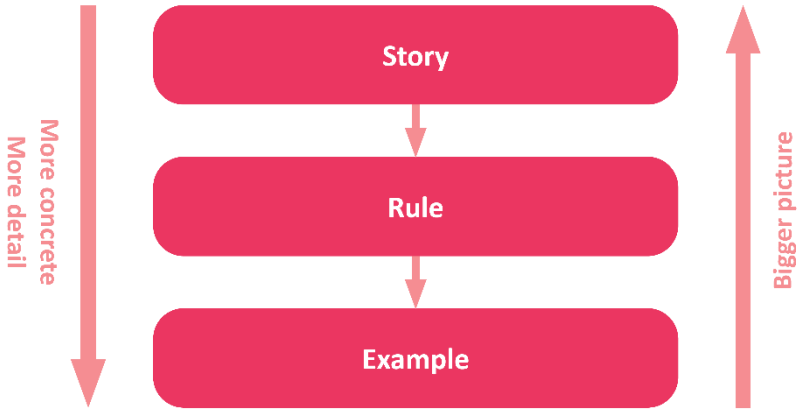


Figure 7 Example Mapping supporting Top-Down and Bottom-Up discovery

In the case that the initial time box is not enough to finalize this process, you may need to look into the three possible reasons for this time management issue:

- The story is too big and needs to be split into smaller stories;
- The story is not clear enough to all participants. There's still room for ambiguity or it's unclear what its purpose is;
- The story has too much uncertainty in it. There's still debate on how this should work and/or if this is actually needed (by the end-user).

Based on the insights, it is recommended to let the product or business representative decide to:

- Extend the current session for a limited time;
- Or stop at the end of the time box and (re)do the homework before the story is planned for an additional Example Mapping session.

Example Mapping sessions provide instant feedback. As a result of (and during) the session, the story is visually being built up on the surface in front of and together with the group. The visual result reflects the current understanding of the story. Typically, the result looks like Figure 8.

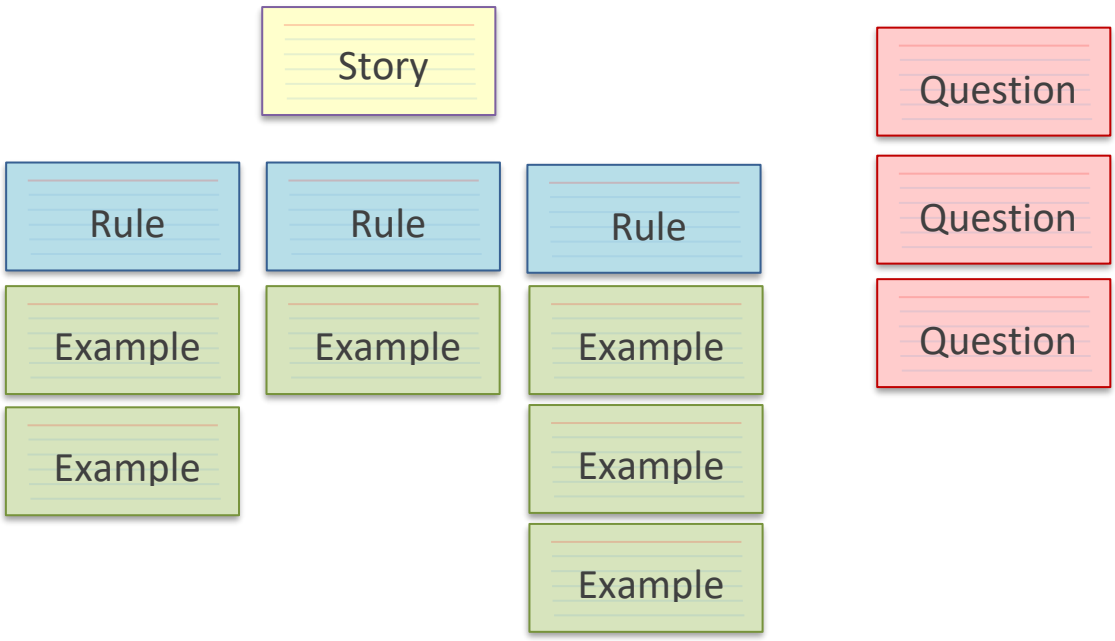


Figure 8 Example Mapping overview

- A surface covered in *red* question cards shows that there is still a lot of unknowns and a lot to learn about this story.
- A surface covered in *blue* rule cards indicates that the story is (too) big and complicated. This story is a candidate for splitting it into smaller stories. These smaller stories result in new *yellow* story cards that can be put on the backlog.
- A rule with many examples is probably too complex or it consists of a combination of different rules. In the case of multiple rules, they need to be discussed and 'exampled' separately.

It is inevitable that during Example Mapping questions pop up or certain assumptions need to be made (for the time being), supported by the red cards for questions (See Figure 8 Example Mapping overview).

Using Specification by Example without Example Mapping poses several risks: examples are by definition very specific and detailed, while the story focuses generally on the big picture. When using Example Mapping, an additional layer can be introduced, using (business) rules to gain a better understanding of the requirements and functionality.

Basically, stories are mostly vague at the beginning of this process, they can be incomplete or inconsistent. This results in inefficient development of user stories, that are larger than expected, less clear than expected, or business representatives or Product Owners who are misunderstood by developers or testers. When this is the case, during Event Storming sessions developers and testers will have a lot of questions, which helps the business representative or Product Owners to make the

examples more specific. The developers and testers will get a better grasp of the ‘bigger picture’. This process helps to create a shared understanding and a common language among all the stakeholders.

Chapter 7 : Specification with Examples

*Specification with Examples (notice the difference with Specification **by** Example) may very well be the most important, but also most challenging, part of CARE. We started with an Impact Map, Event Storm and finally an Example Map. At this point, we will convert the Examples from the Example Map into scenarios, using the Gherkin language (Given-When-Then).*

Keywords
Specification with Examples, Living Documentation, Potentially Shippable Increment (PSI), Specification by Example, Gherkin, Scenarios, Feature Files, Shared Understanding, (the) Goal, Requirements, Three Amigos, Examples

LO-7.1	K1	Recall what Specification with Examples is.
LO-7.2	K2	Summarize why Specification with Examples is beneficial.
LO-7.3	K1	Select which roles are typically involved with Specification with Examples.
LO-7.4	K2	Summarize the stages of Specification with Examples.
LO-7.5	K2	Interpret the goal of Specification with Examples.
LO-7.6	K1	Choose the right equipment needed for Specification with Examples.
LO-7.7	K1	Remember the keywords for Gherkin.
LO-7.8	K3	Apply the Gherkin syntax when writing Specifications with Examples.
LO-7.9	K2	Summarize the rules for Gherkin scenarios.
LO-7.10	K2	Classify how examples can be used for requirements and tests.
LO-7.11	K1	Recall the diagram showing the relations between examples, requirements and tests.

HO-7.1	HO-2	Using Specification with Examples in a team using an example case.

7.1 What is Specification with Examples?

LO-7.1	K1	Recall what Specification with Examples is.

Specification with Examples may very well be considered the core of Behaviour Driven Development and therefore also the core of CARE. All the steps executed up to this point come together in the Specification with Examples, as this step will provide the *living documentation* used for the final steps of delivering a *potentially shippable increment* (working software).

This process is part of *Specification by Example* (see Chapter 5) and consists of writing down the examples that have been specified using Example Mapping (see Chapter 6). The syntax used to write down these examples is called *Gherkin*. It is a structured way of creating examples using natural language, domain language and a limited set of keywords. The examples are written down in *scenarios* which are grouped in *feature files*. More on Gherkin, scenarios, and feature files in paragraph 7.5.

7.2 Why is Specification with Examples useful?

LO-7.2	K2	Summarize why Specification with Examples is beneficial.

As mentioned, Specification with Examples is an essential part of Specification by Example and therefore of CARE as a whole. While it is possible to only use working methods such as Impact Mapping and/or Event Storming and not use Specification by Example, the use of Examples is essential to CARE, as its full name indicates: Capturing Agile Requirements by Example.

The use of concrete examples is a very powerful way to create a shared understanding amongst all stakeholders, leaving no (mentionable) room for misinterpretation or ambiguities (when done correctly). You create a single source of truth with everyone involved and use it for requirements, test and development (ATDD, TDD, see Chapter 8).

7.3 Who is involved with Specification with Examples?

LO-7.3	K1	Select which roles are typically involved with Specification with Examples.
LO-7.4	K2	Summarize the stages of Specification with Examples.
LO-7.5	K2	Interpret the goal of Specification with Examples.

The simple answer to the question “who are involved?” is: “everyone”! Writing down the examples guarantees a *shared understanding* among all of the stakeholders, and allows to refine the defined *goal* into actual, concrete, specific *requirements*. Creating a shared understanding for everyone involved only works when everyone is involved in the process. However, the idea that the entire team and all possible stakeholders will write all the scenarios together is not realistic.

Therefore, it is often decided to divide this process into different stages:

1. Someone writes the first draft of the examples. This is often done by a business analyst, a subject matter expert or, occasionally, a tester who has a deep knowledge of the domain.
2. This first draft is then discussed with a small group of stakeholders, often the *Three Amigos*. The first draft is refined, some elements may be added, removed, or worded differently.
3. Finally, this refined version is presented to everyone involved, both the development team and the involved (business) stakeholders. If everything is clear to everyone (shared understanding) the feature files are finalized (for now). If there are still some elements left unclear, further refinement might be needed. This further refinement may be led collectively by all of the stakeholders, however, if big changes are required, it may also be wise to go back to stage 1 or 2.

Of course, how this is done and who is involved when, is very dependent on the complexity of the new functionality that needs specification. If it is only something small and easy to understand, the Three Amigos stage may be skipped. If it is very complex, then some extra sessions could be added before presenting it to everyone involved.

The ultimate goal is always to obtain a specification that everyone involved completely and fully understands.

7.4 What is needed for Specification with Examples?

LO-7.6	K1	Choose the right equipment needed for Specification with Examples.

The scenarios are written down as plain text, which means any text editor could be used. When creating the scenarios in a group session, it is also an option to use a whiteboard or a flipchart to write them down. However, when starting to work with the scenarios, it is advisable to use some tooling that understands the Gherkin language. Such a tool will highlight certain keywords and it may offer suggestions on re-using parts of examples that have been used earlier in the process. It will also warn you when you do not follow the Gherkin syntax correctly and help you line out text and tables. This can be of great help when trying to create readable scenarios.

Some of these tools can also be used for connecting the scenarios to the test automation framework later on (see Chapter 8). The most well-known tools are Cucumber (https://cucumber.io/) and SpecFlow (https://specflow.org/), but there are many more.

7.5 How does Specification with Examples work?

LO-7.7	K1	Remember the keywords for Gherkin.
LO-7.8	K3	Apply the Gherkin syntax when writing Specifications with Examples.
LO-7.9	K2	Summarize the rules for Gherkin scenarios.
LO-7.10	K2	Classify how examples can be used for requirements and tests.
LO-7.11	K1	Recall the diagram showing the relations between examples, requirements and tests.

HO-7.1	HO-2	Using Specification with Examples in a team using an example case.

Examples are written down in scenarios which are placed in so-called feature files. These are plain text files. It is advisable to not only write the scenarios down in this feature file, but also the story and rules that were discussed during Example Mapping. In a later stage, it is also possible to add tests to the same feature file, as long as a clear distinction is drawn between what is specification and what is test.

The most important aspect of creating this feature file is to write the actual scenarios using the Gherkin syntax, with the three keywords ‘Given’, ‘When’ and ‘Then’:

Given the following prerequisites
When the end user performs this behavior
Then the system has responded with that behavior

This is the foundation of Gherkin and Specification with Examples. There are some additional keywords that need to be used:

- ‘Scenario’: Used as the start of a new scenario and followed by the name of the example (as was chosen during the Example Mapping session)
- ‘Scenario Outline’: Same as scenario, but with the option to add parameters in your example to turn it into more similar, parameterized scenarios, using data tables which are made using pipes (|) for columns.
- ‘And’: Can be used to add extra rules of ‘Given’s, ‘When’s and ‘Then’s.
- ‘But’: Can be used as an alternative keyword in scenarios. Not advisable, because it usually leaves room for unnecessary complexity and therefore misunderstandings.

For example:

```
Scenario: A professional successfully signs up for a training course.

Given a training course 'CARE-jul-21' has been planned for '12-07-2021'
And the training course 'CARE-jul-21' is not yet fully booked
When professional 'Jan' registers for the 'CARE-jul-21' training course on '10-07-2021'
Then 'Jan' is registered as participant for the training course 'CARE-jul-2021'
And 'Jan' has received the option to save the date '12-07-2021' in his calendar
```

There is a **Scenario** name that clearly describes which functionality is specified. It is a clear description, that leaves no room for misunderstandings, and it also states clearly why this scenario exists and what its purpose is.
The **Given** states all the prerequisites that are relevant or required to make this scenario work. Note that there are two parts in the Given segment, the second one uses the keyword **And**.
The **When** describes what an end user actually does, what the exact behavior is.
Finally, the **Then** describes the required response of the system, again using the keyword **And** to indicate multiple parts. It is a good habit to write this in present perfect (or, if preferred, past perfect) tense to indicate that the action of the system is completed and what is described is the present result of these events.

Now, if you would want to create a few alternative but similar scenarios, you could use a Scenario Outline, for example:

```
Scenario outline: A professional signs up for a training course.

Given a training course 'CARE-jul-21' has been planned for '12-07-2021'
And the training course 'CARE-jul-21' is <booked>
And a training course 'CARE-aug-21' has been planned for '12-08-2021'
And the training course 'CARE-aug-21' is not yet fully booked
When professional 'Jan' registers for the 'CARE-jul-21' training course on '10-07-2021'
Then 'Jan' <registered> as participant for the training course 'CARE-jul-21'
And 'Jan' has received the option to < option>

| booked                | registered          | option                                          |
| not yet fully booked  | is registered       | save the date '12-07-2021' in his calendar      |
| fully booked          | is not registered   | register for training course 'CARE-aug-21'      |
```

When writing examples, there are several things to always keep in mind. **The focus should always be on user behavior, not on implementation**. The user behavior is what is actually changing and what will lead to an event, which usually is a system responding to the behavior. The actual behavior of an end user is always described in the When section and is required. Pre-requisites do not describe the behavior of an end user and are therefore not always required in a scenario. Also note that an end user can be either a person or another system (see also the actors from the Impact Map, Chapter 3).

Examples should be realistic and should preferably use real-life data. You want to describe what will actually happen in the real world, so do not use unrealistic examples that will never occur, unless they create a better understanding of the system. In that case, it is good to add a remark to the

scenario stating it is a hypothetical scenario, just to clarify what would happen in that unlikely situation.
Use specific answers and refrain from using 'yes/no' answers. It is better to say that, for instance, a specific error will be shown (e.g., "User input invalid") instead of simply saying there will be "an error message". Whenever you can still ask further questions about the scenario, then it is not concrete enough.
Always be concrete, not abstract. That is why, in the previous example, there isn't spoken about just any professional, but about a professional with the name 'Jan'. By doing so, we can also make it very clear that not just any professional is registered, but the actual same professional that registered himself for the training course. Again, there should be no room for additional questions. If this example had indicated "a professional", you would still have to ask, 'Which professional?' It is also possible to use personas to render the examples even more specific when needed. We also do not use ranges in data, so instead of saying that an actor is between 18 and 35 years old, you would say the actor is 25 years old. The range can still be described, but this should be done in the scenario title or the (business) rules.

Never make any assumptions. Always make sure that what you are saying is what the business actually wants. When in doubt: ask people who know. Keep in mind that you are writing the requirements for the system to be built and these requirements will also be the basis for tests run to verify whether the system was built correctly. Any mistakes made at this stage can therefore have huge consequences later on during development of the system, or even lead to incidents in production.

Do not add tests. The scenarios may look like tests and will eventually be used as tests (and might drive test automation, more on that in Chapter 8), but Specification with Examples is about specification, not about testing!

Finally, always look for implied concepts and side effects. Are there issues that could also happen but which you did not specify? Then specify them. Are there certain side effects that could cause unwanted behavior from the system, possibly contradicting other scenarios/specifications, then change the scenario to prevent this from happening?

When the examples are specified correctly, they elaborate on the requirements. The examples can become tests (but keep in mind: specification first, test second) and these tests verify the implemented requirements. By doing so, everything is connected with everything, as the diagram in Figure 9 shows.

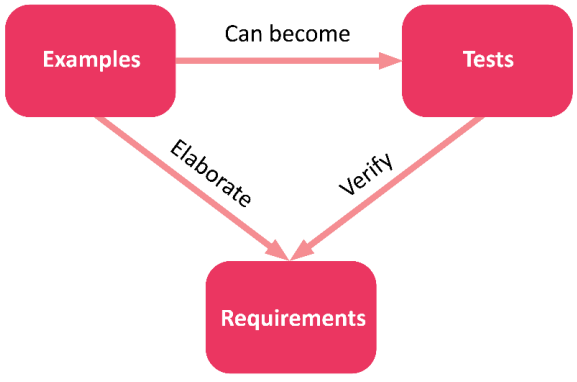


Figure 9: Relationship between examples, tests and requirements

Chapter 8 : (Acceptance) Test Driven Development (introduction to Test Automation using BDD)

This chapter will define (A)TDD, its process and viewpoints and the use of automation. ATDD is not really a part of the specification process, but it is one of the aspects that make CARE and BDD a very powerful way of working. It makes it possible to directly link your tests and test automation to your specification/requirements. Consequently, you can obtain executable specification that is linked to the test automation and are therefore linked to the (production) code, creating one single source of truth.

Keywords

Acceptance Test Driven Development (ATDD), Test Driven Development (TDD), Acceptance Test, Development Test, Short Feedback loops, Test Automation, Layers of Automation, Automation Pyramid, Business-faced, Developer-faced, TRIMS

LO-8.1	K1	Recall what ATDD is.
LO-8.2	K1	Remember what an Acceptance Test is.
LO-8.3	K1	Recall what TDD is.
LO-8.4	K1	Remember what a Development Test is.
LO-8.5	K2	Compare ATDD and TDD and explain the differences.
LO-8.6	K2	Summarize how ATDD and TDD work together.
LO-8.7	K2	Summarize why you may need (A)TDD.
LO-8.8	K1	Define which roles are involved for (A)TDD.
LO-8.9	K1	Recall the types of tools that are helpful for (A)TDD.
LO-8.10	K1	Remember that introducing a tool is not a goal by itself but should support goal(s).
LO-8.11	K1	Remember the rationale of the syntax for an Acceptance Test.
LO-8.12	K1	Recall the pitfall of the syntax for an Acceptance Test.
LO-8.13	K3	Use the syntax to write (Acceptance) Test(s) based on the specifications with examples.
LO-8.14	K2	Summarize the process steps of (A)TDD.
LO-8.15	K1	Remember the layers of automation.
LO-8.16	K2	Classify text/code fragments to the layers of automation.
LO-8.17	K1	Remember the layers of the Test Automation Pyramid.
LO-8.18	K1	Recall the main considerations for the use of automation for/of tests.
LO-8.19	K2	Classify if automation is proper according to TRIMS.

8.1 What is (A)TDD?

LO-8.1	K1	Recall what ATDD is.
LO-8.2	K1	Remember what an Acceptance Test is.
LO-8.3	K1	Recall what TDD is.
LO-8.4	K1	Remember what a Development Test is.
LO-8.5	K2	Compare ATDD and TDD and explain the differences.
LO-8.6	K2	Summarize how ATDD and TDD work together.

Acceptance Test Driven Development can be seen as an extension of *Test Driven Development* that further includes business rules and functionality. It aims to help you set your goals first, and then work towards those goals.

The definition of Acceptance Test Driven Development is:

"*Acceptance Test Driven Development (ATDD) involves team members with different perspectives (customer, development, testing) collaborating to write acceptance tests in advance of implementing the corresponding functionality.*" (AgileAlliance.org, 2021)

For Test Driven Development the definition is:

"*Test Driven Development (TDD) refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).*" (Agile Alliance, 2021)

(A)TDD is designed to drive development using (acceptance) tests, like in TDD to which it is closely related. Both ATDD and TDD dictate that one creates tests first and then starts writing the code to build the application. The focus of TDD is on unit tests and integration tests that help writing maintainable code, while the emphasis of ATDD is on the collaboration between developer, tester and business. (A)TDD encompasses acceptance testing, but highlights writing the acceptance tests before developers begin creating unit tests and start coding.

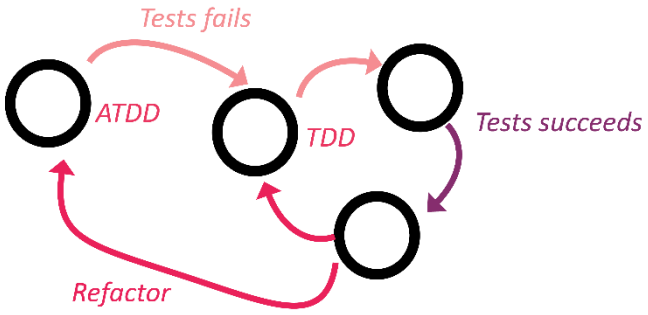


Figure 10: (Acceptance) Test Driven Development

Acceptance tests are created from the end user's point of view: an external (black-box) view on the system. They examine externally visible effects, such as specifying the correct output of a system given a particular input: the (system) behavior. Acceptance tests can verify how the state of an element changes, such as an order that goes from status 'paid' to 'shipped'. They can also check the interactions with interfaces to other systems, such as shared databases or web services. In general, they are independent from the implementation and do not require automation, although the

automation of these tests will help with regression testing and most likely will be dependent on the implementation.

More tests may be needed to support the development of the required functionality (e.g., unit and integration tests or *Development tests*) and to ensure code that is easy to maintain. This is (development) TDD, which requires test automation. These tests can often be derived from the acceptance tests since the units implement some portion of a requirement. ATDD tests should be readable by the customer; TDD tests do not need to be.

Test Driven Development practices are much better documented and understood in the software development community than Acceptance Test Driven Development. So, if a team or organization already has good TDD practices in place, there is probably no need to demonstrate the value of automated tests or to change the design to make their software more testable.

8.2 History of (A)TDD

Background reading

The principles of Test Driven development technique date back to NASA's early 1960's 'Project Mercury'. It was (one of) the first software project(s) using Test Driven Development and other agile practices.

"Project Mercury ran with very short (half-day) iterations that were time boxed. The development team conducted a technical review of all changes, and, interestingly, applied the Extreme Programming practice of test-first development, planning and writing tests before each micro-increment." (Larman & Basili, Iterative and Incremental Development: A Brief History, 2003, p. 1)

The term [Test Driven development] was devised by Bertrand Meyer and first described in various articles starting in 1986." (Meyer, 1986)

Kent Beck "helped pioneer the rediscovery of 'test-first programming'". The 'test-first programming' (TP) is one of the concepts of extreme programming, defined in his book 'Extreme Programming (XP): Explained' (October 1999), later followed by a book 'Test Driven Development By Example' in which he mentions ATDD briefly, but dismisses it as impractical. Despite Beck's objections and driven by the popularity of tools such as Fit/FitNess, ATDD has become an accepted practice.

8.3 Why is (A)TDD useful?

LO-8.7	K2	Summarize why you would need (A)TDD

The main reason for working with ATDD is to improve communication, obtain clear requirements and realize them during the entire value stream. Applied correctly, this should result in building exactly what was asked for and, ultimately, in making the customer happy. Important reasons for working with ATDD are:

COLLABORATION

As it is the case in all of the methods presented within CARE, collaboration plays a key role here again. ATDD using BDD encourages the collaboration of all the parties involved. By collaborating, having a direct conversation with people who are responsible for the specification and the validation of the behavior, any misunderstandings are discovered and resolved quickly. Having developers

involved in the discussions about the requirements provides them with insights and the reasoning behind the requirements. The implementation of a requirement should not be part of the discussion.

AGREEMENT ON REQUIREMENTS

As the business has collaborated with development to define the acceptance criteria for the user story (requirement), if the test passes, the feature is deemed functionally acceptable to the business. The acceptance tests formalize a consensus, while they remain flexible for future insights and refactoring. Instead of fixed documentation, the agreement is to discover what is needed throughout the lifespan of the product.

CLEAR ACCEPTANCE CRITERIA

Clarity requires discovering ambiguities which often lie in the area of "you don't know what you don't know." Although ATDD using BDD offers no guarantees, the interactions between the different roles and the disciplined approach it provides participate greatly to creating clarity (shared understanding).

REDUCED TIME FOR FEEDBACK

BDD enables a *short feedback loop* by discussing acceptance criteria together with the requirements, with the people who care about them or are at stake. This allows for misunderstandings to be often found and resolved immediately. This is critical for improving flow of value and reducing delays.

IMPROVED DEVELOPER / TESTER COMMUNICATION

In practice, there is often a natural rivalry between developers and testers. The testers are usually the people bringing the 'bad news' when something is wrong. By improving the communication and involving them up front, then both testers and developers have a better understanding, and also it is usually possible for testers to begin writing tests at the same time as the developers start writing their code.

IMPROVED PRODUCT QUALITY

Tests and requirements (or specifications) are interrelated. A requirement that lacks a test may not be implemented properly. A test that does not refer to a requirement is unneeded, or it indicates a missing requirement! An acceptance test that is developed after the implementation began, represents a new requirement. If requirements and tests are in sync, then the right software will be built correctly, and, as a result, the overall product quality will improve.

PREPARED FOR AUTOMATED TESTING

BDD does not require automation, but it makes it easier by defining acceptance criteria in a form that can be easily automated. If the developers are into TDD (which requires automation) and they realize the time spent for ATDD is dedicated to defining tests and not to automating them, they are very often open to assisting with the test automation of the acceptance tests.

PRODUCTIVITY / FOCUS

Another key benefit is to have clearer goals or targets for development and to prevent functional regression. The tests are written before any code is written. The developers implement the system using the (acceptance) tests. After each 'commit', the developers can check how (well) they are doing by running the tests. Failing tests provide quick feedback that the requirements are not yet being met.

IMPROVED CODE QUALITY

For developers, one of the most important design skills is to "consider your tests before writing your code". This is consistent with the first mantra of Design Patterns: "Design to the behavior of your classes before considering how to implement them." (A)TDD's formulation of tests, prior to code, drives design. High quality code is easier to test. The reverse is also true. Code that is easy to test is of higher quality than code that is not. The Test first process dictates that you define your tests before you write code, which improves your design.

8.4 Who is involved in (A)TDD?

LO-8.8	K1	Define which roles are involved in (A)TDD.

Simply put: everyone. All the roles associated with the project should be represented by some persons. ATDD is a development methodology based on communication & collaboration between the business customers, the developers, and the testers. There may be other stakeholders or roles that may also need to be involved.

The (acceptance) tests are specified in business domain terms. The terms then form a ubiquitous language that is shared between the business, developers, and testers.

8.5 What is needed for (A)TDD?

LO-8.9	K1	Recall the types of tools that are helpful for (A)TDD.
LO-8.10	K1	Remember that introducing a tool is not a goal by itself but should support goal(s).

Firstly, it should be stated that (A)TDD is not a 'one size fits all' solution. It all depends on the context of your organization, project and the people working on the project.

Most of the elements needed for 'Specification with Examples' (chapter 7.4) will also be helpful for (A)TDD as, similarly to Specification with Examples, it starts with writing the tests in Gherkin.

(A)TDD consists in transforming requirements into tests. All the tests mentioned should be a part of an overall testing strategy, which should be kept in mind at all time. Even if all scenarios are written, and tests have been derived from the requirements, having a testing strategy to ensure that all areas of risk are being covered is still mandatory. For (A)TDD specifically, it is important to consider two types of tests: Acceptance Tests and Development Tests. There are various types of tools that are helpful – and possibly even required.

For Development Tests, automation is required. So, any xUnit testing framework, in line with the technology stack of the SUT, will be useful.

Types of tools for TDD:

- (Unit) test execution (required)
- Generic automation framework
- Test reporting

For the Acceptance Tests, automation is not necessary, but tools like Cucumber and SpecFlow can help automate these tests later on. This is a great way to work according to Acceptance Test Driven Development.

Types of tools for ATDD:

- Translation framework
- Generic automation framework
- Test reporting
- Test execution

Tools are tools; they should help achieve a result. Introducing a tool is not a goal by itself, it should rather support your goal(s). Start by defining your goals and requirements, but also look at the competences within your project to see whether they match with any tools on the market (or if they don't: you may also want to consider developing your own tools). Also consider policies within your project/company, which might prescribe certain (type of) tools or impose any restrictions on the use of tools. Be aware that you should treat the introduction of a tool as a project.

8.6 How does (A)TDD work?

LO-8.11	K1	Remember the rationale of the syntax for an Acceptance Test.
LO-8.12	K1	Recall the pitfall of the syntax for an Acceptance Test.
LO-8.13	K3	Use the syntax to write (Acceptance) Test(s) based on the specifications with examples.

When the specifications have been illustrated with examples, as mentioned in chapter 7 'Specification with Examples', this step offers us a good starting point for (A)TDD. These scenarios are the acceptance criteria; as they are a description of what could be checked by a test. Given a requirement such as "As a professional, I want to sign up for a training course from the training calendar", an acceptance criterion might be, "verify the student is registered as participant for the training course". An acceptance test for this requirement further refines the criterion so that the test can be run with the same result each time.

For acceptance tests, we usually follow the following format (which is, obviously, the same as with Specification with Examples):

- Given (setup) – A specified state of a system.
- When (trigger) – An action or event occurs.
- Then (verification) – The state of the system has changed, or an output has been produced.
- And – It is possible to add statements in any of the other sections (given, when, then).
  Be very careful when using 'And'! Using many 'Ands' should warn you about the size of the scope of your (test)scenario.

An example of a specification/acceptance test for the mentioned acceptance criterion:

Scenario: A professional successfully signs up for a training course.

Given a training course 'CARE-jul-21' has been planned for '12-07-2021'
And the training course 'CARE-jul-21' is not yet fully booked
When professional 'Jan' registers for the 'CARE-jul-21' training course on '10-07-2021'
Then 'Jan' is registered as participant for the training course 'CARE-jul-2021'
And 'Jan' has received the option to save the date '12-07-2021' in his calendar

More tests may be needed, for various reasons, mainly to make sure this functionality is tested well. Also, some technical tests may be written/needed (e.g., unit tests) to drive easy-to-maintain code. They can also be written down in the same format as the scenarios they are based on. For example, this second test scenario could be written:

Scenario: A professional cancels the registration for a training course.

Given a training course 'CARE-jul-21' has been planned for '12-07-2021'
And the training course 'CARE-jul-21' is not yet fully booked
When professional 'Jan' cancels the registration process for the 'CARE-jul-21' training course
Then 'Jan' is not registered as participant for the training course
And a user message is shown 'Registration was cancelled'

PROCESS FOR (A)TDD

LO-8.14	K2	Summarize the process steps of (A)TDD.

When the specifications have been made clear are agreed upon, and acceptance tests have been written, the first acceptance test is added to the suite. The next step is to run the test – most likely, it will fail. If, for some reason, the test passes, another acceptance test shall be added to the suite and then the two tests can be run. However, again it is expected that the test(s) will fail, thus the system requires minor changes. These changes might be: configuration management for the system, for example a new version of the software that needs to be deployed, or some settings that need changing. But usually, these changes are code changes, required to add/change the desired functionality that you are testing with the acceptance test(s). At this point, Development TDD starts.

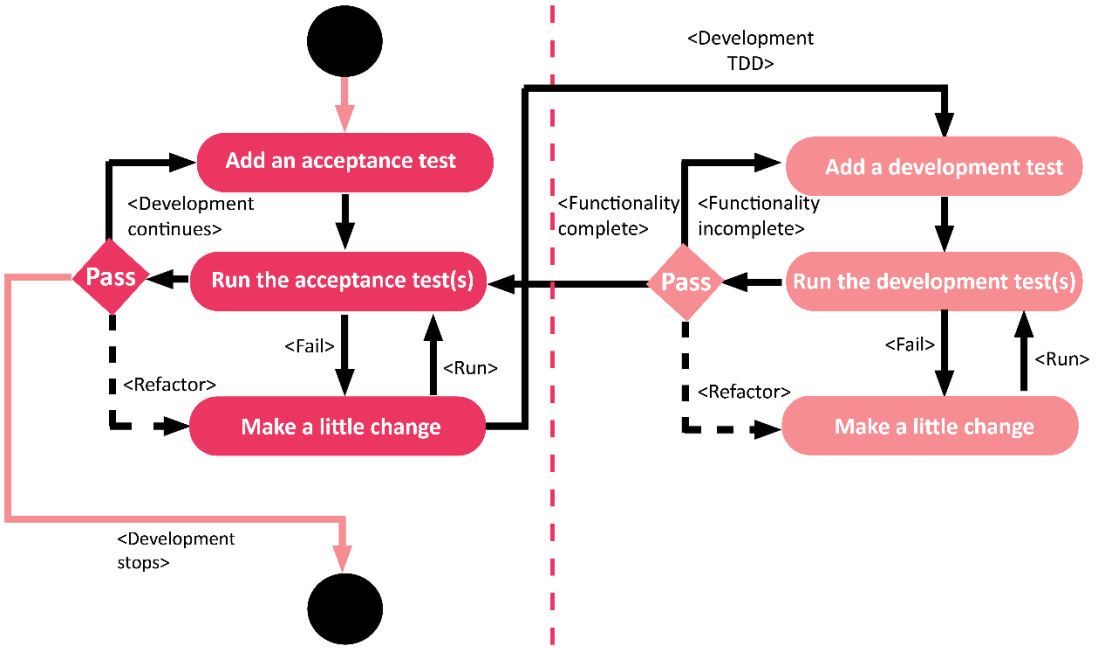


Figure 11: Sequence diagram for (A)TDD

Development (unit) tests are derived from the acceptance test(s) and added (one-by-one) to the TDD suite by the development team. Developers only start implementing the production code after

running the test(s) first. If it does pass (e.g., there already was production code) a new test can be added to the suite. Otherwise, the code is written/updated to make it pass the new test(s). The fourth step is to run the tests again. If they fail, you need to update your code and retest until the tests pass. It is important to keep focused on making each single test succeed with just enough production code to fulfil that test, so nothing more and nothing less.

Once the tests pass, the next step is to repeat the process: Add new development tests until the team believes no further new tests are needed, meaning the functionality is complete. When all the tests go 'green', then refactoring can be done. The inelegant code that had been written to make the tests succeed at first is now honed in. This might make the tests fail again, and the process is repeated until (all) the tests pass.

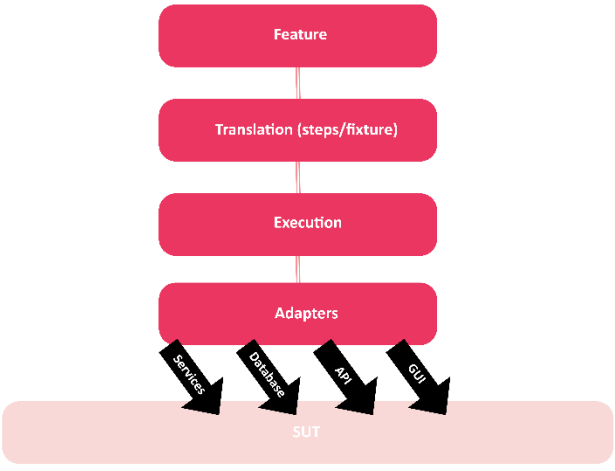
Eventually, when all the development tests pass, the acceptance test(s) are ran to see if they also pass. This again might require some rework (refactoring) as to implement the new insights and or new development tests that have been written. If there are no more acceptance tests for a particular user story, then the following test can be added. A user story is considered done when all acceptance tests are succeeding. At this point, the development stops and the (A)TDD process comes to an end.

Layers of Automation

LO-8.15	K1	Remember the layers of automation.
LO-8.16	K2	Classify text/code fragments to the layers of automation.

Using the Given-When-Then format to define your ATDD and TDD tests makes them suitable for automation. There are many tools that can accommodate this, in general they all require some layers of automation.

On the top level (Feature), the tests are written in the 'Gherkin' 'Given-When-Then' format. It typically uses a set of special keywords to give structure and meaning to executable specifications. Tests are housed in 'Features', which is a text file with a '.feature' extension. Each feature consists of one or more 'Scenarios', which are the test cases. Various scenarios are needed to test a certain functionality and are usually grouped in one 'feature-file'. Then the scenarios are made up of 'Steps', which is the component interactions for a given scenario or test case. Both acceptance tests and development tests can be defined on this level. It provides a readable (text) format for all the stakeholders, with some structure to make it automatable.



F

Figure 12: Test Automation Framework

The 'Translation' layer, also known as 'steps / fixture' or 'Step Definitions', contains a step-by-step procedure which will make it possible to convert the simple text 'Steps' from the feature file into code that can be executed. This is strongly dependent on the technology / language used for the 'System Under Test' (SUT). This layer is also called 'glue code' because it glues the translated tests written in text to the executable code.

In the 'Execution' layer, most of the code found is to make the framework actually execute the tests. It also connects to other code that you need, for example, for writing logs to a log file, and gathers the results of the tests into some dashboard to review the results after a test run.

'Adapters' provide the ways of communication with your 'System Under Test' (SUT). They are called by the execution layer and enable access to the control (e.g., performing actions on the SUT) and observability (e.g., insight into the behavior of the SUT) of the SUT. The most common adapters are shown in the visual representation of this model. They are: the Graphical User Interface, the Application Programming Interfaces (APIs), the database and Services.

AUTOMATION 'PYRAMID'

LO-8.17	K1	Remember the layers of the Test Automation Pyramid.

For the automation of tests, there is a lot to consider, including the level on which the test is, or can be, performed, and with which level of isolation. The best-known visualization of these levels is described by the 'Test Automation Pyramid' created by Mike Cohn (Cohn, 2009). In this model, three levels of isolation of tests are defined. It also specifies the number of tests that should be done on each level, for multiple reasons.

On the base level of this figure, the lowest level and most isolated way of testing, we find 'unit testing'. Tests on this level are very specific, not only in what they test, but also in what they tell the programmers when they fail. Because the scope of unit tests is very small, it will report on which line in the code the bug was found, which makes narrowing down what might be the cause of the bug much easier. Unit testing is often described as the foundation of a solid test automation strategy and therefore represents the largest part of the pyramid.

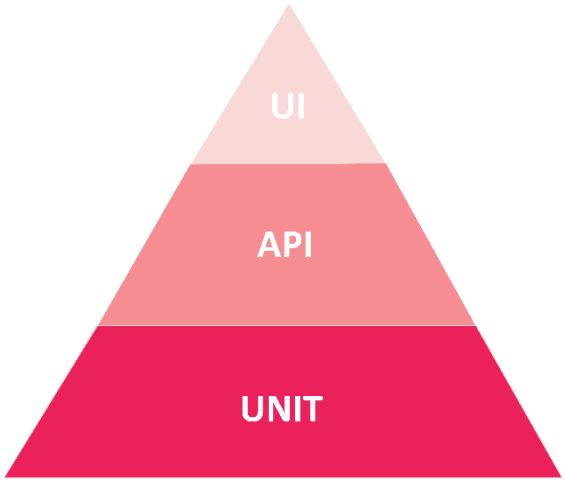


Figure 13: Test automation pyramid

At the top of the pyramid, we find the tests on the (graphical) user interface (UI). In general, we want to run as few of these tests as possible, as they are costly to develop and maintain, expensive to run and they are often brittle. In addition, testing via the user interface is partially redundant, because to get to 'the guts' of the application, you will mostly use the same parts of the user interface multiple times. Therefore, testing this way should be kept to a minimum and, although you might need to run lots of tests, not all of them need to be run through the user interface. In most cases, there is a way to neutralize the UI and get past it using the 'middle layer' of an application, often called 'API'. An API

(or service) provides a gateway to access (more) directly to the core of the application. It accepts some input or a set of inputs and, as a result, the system will process them and give a response. As a consequence, the UI is not needed for testing, and it is still possible or even easier to test the application automated and with less effort and faster feedback.

Audience & Purpose

A drawback of the 'traditional' test pyramid of Mike Cohn is that, in practice, it creates a clear separation between the testing done in the technical layers (Unit / API) and in the more business oriented layer (UI). It is important to distinguish the purpose and audience of the tests, regardless of the level of isolation on which it is (or can be) executed. On the one hand, there are tests of interest to the business (business-faced), which answers the question: "Are we building the right 'system'?" And on the other hand, developer-faced test, needed for the development of the functionality, which answers the question: "Are we building 'system' right?" Both business-faced and developer-faced tests can be executed on various levels of isolation, and so both could have their 'own pyramid'. This also means that unit tests are not only of interest to developers, and in theory, a test on the UI can also be of interest to the developers. It is important to communicate about both tests, to avoid any redundancy of testing.

To automate or not to automate?

LO-8.18	K1	Recall the main considerations for the use of automation for/of tests.
LO-8.19	K2	Classify automation as proper or not, according to TRIMS.

Automating your acceptance tests and development tests is viewed by many companies as a holy grail. However, the question is whether this is a good strategy. to obtain functioning TDD, it is essential to have development tests automated. For acceptance tests, it is not necessary, although automation will help with regression testing. This means there are some considerations to be made before you start automating everything. The three main considerations to make regard: technical challenges, deadline(s) and costs.

Automated regression set versus Temporary

The first consideration is to determine whether the tests are for temporary use during development, or they are to be considered valuable for regression testing. An example of a temporary test would be a test written to double check whether the results from testing are correct. If the 'regular' tests appear to be trustworthy, they can become part of the regression set and the temporary test can be archived. A temporary test might still be automated if it is easy to write but may very well not be worth the (automation) investment.

Automated regression set versus Manual

Tests that need to be executed repeatedly are best not executed manually. They are most certainly suited for automation. Some tests are easier to run with automation, for example a load test with 100 users logging in and registering for a training course within a short period of time. Doing this manually will mean you need to click through the UI of the application 100 times. Even if you only check this just occasionally, it will still make sense to automate this test, as doing the same thing 100 times is labor intensive and even boring and error prone too.

Properly Automated?

When performing test automation, there are many guidelines you can use that will help you 'properly' automate your tests. Most of them are summarized by many existing acronyms, one of which is called TRIMS (Bradshaw, 2019):

- Targeted – Is the test targeted to a specific risk and automated on the lowest layer the testability allows?
- Reliable – Is the test deterministic, to maximize its value and to avoid flakiness (tests that sometimes fail and sometimes pass, even though there are no changes in the System, nor in the test automation framework)?
- Informative – Does the test provide as much information as possible to aid further analysis of test results, both when passing and failing?
- Maintainable – Does the test have a high level of maintainability to be able to keep up with constant changes?
- Speedy – Are the execution and maintenance of the test as fast as the testability allows, so rapid feedback loops can be achieved?

Automation is more than installing a tool and should be treated as a project with a clear strategy, clear goals and clear and feasible expectations.

Chapter 9 : How to start with CARE?

Now that you know what CARE is, you can start using it in the real world. This may be challenging, however, even now you know the benefits and pitfalls. You'll need a clear goal and purpose, to realize the importance of psychology in both individuals and teams, and to create the right mindset and eventually plan a pilot project.

Keywords
Teamwork, Psychology, Mindset, Pilot project

LO-9.1	K2	Demonstrate the key benefits of CARE using the questions Why?, Who?, How? and What?
LO-9.2	K2	Understand teamwork and psychology so you can start using CARE successfully.
LO-9.3	K1	Recall what should be in place for a pilot project according to the principles of CARE.

9.1 Key benefits

LO-9.1	K2	Demonstrate the key benefits of CARE using the questions Why?, Who?, How? and What?

When starting with CARE, it is important to know what the key benefits (and limitations) are and how to communicate them to the involved team(s). These key benefits can be discovered by asking the four questions we know from Impact Mapping (see ChapterChapter 3):

Why would we want to introduce CARE?
Who should be involved?
How are we going to change people's behavior?
What do we need for that?

Some possible answers to these questions are listed below. But please be aware that the actual answers will be very specific to every individual situation, team, project, organization.

WHY WOULD WE WANT TO INTRODUCE CARE?

If you want to be more in control and to deliver software an end user really cares about, then CARE may be a solution. It offers good working methods to elicit requirements and to and write specifications which are directly linked to the tests and, therefore, to the actual system to be delivered, making everything connected. That means a single source of truth with the purpose to create a shared understanding and the goal to deliver a potentially shippable increment that actually fulfils the end-users' and other stakeholders' requirements. Nothing more and nothing less.

WHO SHOULD BE INVOLVED?

The simple answer to this is: everyone! However, this is often not realistic in the beginning. It is good practice to make a strategy on how to gradually introduce CARE. It is always best to start with CARE right from the beginning, when a new project is started and/or when a team starts with a (more) Agile way of working. However, it is also perfectly possible to turn an existing project into a CARE project. It is often a good idea to start small, e.g., with a pilot, in which one piece of functionality or one team is chosen to try out the CARE way of working and experience the CARE mindset. Depending

on the situation, this can be done rather free-format or using a strict process (which will be loosened later on). It should be made very clear in the beginning who is to be involved and what is expected of everyone. The Three Amigos could start the CARE adventure, gradually taking everyone else along on the way…

How are we going to change people's behaviour?

For instance, a Business Analyst starts with drawing the first impact map together with one developer and one tester. They together try to find a common language which is used to write down the first story and accompanying (business) rules (example mapping). The story and rules are then turned into the first concept feature file that is presented to the entire development team and other stakeholders. By doing so, you are creating an oil slick that slowly spreads throughout the project and possibly the entire organization. Besides some training and possibly coaching, it is most important to start creating a mindset and to define a common language. As is often the case, the proof of the pudding is in the eating and practice makes perfect. When a common language is set from the start, and everyone has a first taste of how the feature files are going to look, processes like Event Storming can be added (which require input from everyone). Some thought should be put in creating a test automation architecture to connect to the feature files as well as in creating a structure in which to organize the now growing number of feature files in order to make everything properly scalable.

What do we need for that?

Finally it is time to find out what is needed to make this new way of working work. Most of the working methods do not require expensive or complex tools. Think more about training people, maybe hiring a coach, but also make sure the right tools get selected to support CARE/BDD and, of course, reserve the needed time to give everyone the opportunity to learn, discover new skills and possibilities (as well as possibly some limitations) and to create the right mindset.

9.2 Teamwork and psychology

LO-9.2	K2	Understand teamwork and psychology so you can start using CARE successfully.

Introducing CARE is all about change management. Changing things is always (somewhat) scary to people and we can lessen this by introducing change more gradually, and by finding the drive to change. Make sure everyone involved understands why change is needed so our brain and we, ourselves, can get used to things changing without feeling threatened by it. In that perspective, it is important to understand what the *goal* (Goldratt, 1984) and purpose of the change is: Why do we want to introduce this new way of working? What do we want to achieve? How will we do that? And possibly the most important question: When will we be happy? When are we done?

It is important to involve everyone in this change. Teamwork is essential and there are many "I"s in team! A team consists of individuals, and they need to work together. If even one of the team members is not 'on board', this could prove disastrous for introducing CARE successfully. It's therefore important to consider everyone's (potential) specialties/abilities, everyone's personalities, everyone's (possible) limitations and make change happen together by creating the right mindset. A mindset of continuous communication, continuous agility and in which everyone involved actually feels involved and knows the goal and purpose of what they are doing, and in which creating shared understanding is always on everyone's mind.

Creating this mindset starts with making sure everyone knows what CARE is and what the key benefits are (see section 9.1). The teams and team members who will be working with CARE need to

be ‘on board’ and everyone should be willing to change. Once this step has been achieved, it is time to just start working with CARE, starting small, e.g., by beginning with a pilot project.

9.3 Starting a pilot project

LO-9.3	K1	Recall what should be in place for a pilot project according to the principles of CARE.

The best way to implement CARE into a project or organization is to just start. It can thus be wise to choose a pilot project first (possibly first with just a ‘light’ version of CARE) and let it grow from there, at least if the pilot turns out to be successful. A pilot project should be a project that is complex enough, yet it should not be too critical to the organization: there should be room for failure… Failure is part of the learning process.

It is important to make sure everyone involved knows about the pilot project and knows what is expected of them. They should be aware that it is an experiment in which they are allowed to make mistakes. They should feel safe, there should be trust in the teams involved and the teams should be made responsible for the project (if they are not already) and feel responsible for it. There should also be enough time reserved for the teams to learn this new way of working and create the required mindset. That means a pilot project should preferably not have a tight deadline.

The next step is starting with CARE and making sure to deliver software an end user really CAREs about.

Delivering software an end user really CAREs about

CARE: Capturing Agile Requirements by Example. Consists of using the right working methods, tools, stimulating the right mindset, and introducing a way of working that allows us to create a shared understanding of why we need to deliver certain software, who it is aimed for, how it should impact these actors and what we need to do to obtain these results. To achieve this, we can use tools and ways of working such as Impact Mapping, Event Storming, Example Mapping, Specification by Example, (A)TDD practices and more. In the end, we care most about changing the end-users' behavior in such a way that it makes their life easier, better, faster, cheaper, etcetera. The goal and purpose of CARE is after all to create software an end user really ***CARE****s about.*

References

Specific references

Abrial, J.-R. (1988). *The B tool (Abstract).* Springer-Verlag Berlin Heidelberg.

Adzic, G. (2012). *Impact Mapping.* Surrey: Provoking Thoughts Limited.

Adzic, G. (2012). *Specification by Example.* Shelter Island: Manning.

Adzic, G. (2021, 05 14). *Impact Mapping Website*. Retrieved from Impact Mapping Website: http://www.impactmapping.org/

Agile Alliance. (2021). *TDD*. Retrieved from agilealliance.org: https://www.agilealliance.org/glossary/tdd/

Agile Manifesto. (2001). *Agile Manifesto*. Retrieved from Agile Manifesto: https://agilemanifesto.org/

AgileAlliance.org. (2021). *Acceptance Test Driven Development (ATDD)*. Retrieved from agilealliance.org: https://www.agilealliance.org/glossary/atdd/

Anderson, L. K. (2022, 03 15). *REVISED Blooms Taxonomy Action Verbs*. Retrieved from https://www.apu.edu/live_data/files/333/blooms_taxonomy_action_verbs.pdf

Anderson, L., Airasian, P., & Krathwohl, D. (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Allyn & Bacon.

Bach, J. (n.d.). *https://www.satisfice.com/download/heuristic-test-strategy-model*. Retrieved from Satisfice.

Bradshaw, R. (2019, August 5). *TRIMS - A mnemonic for valuable automation in testing.* Retrieved from automationintesting.com: https://automationintesting.com/2019/08/trims-automation-in-testing-strategy.html

Brandolini, A. (2013, November 18). *Introducing Event Storming.* Retrieved from ziobrando.blogspot.com: http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html

Cohn, M. (2009, December 17). *The Forgotten Layer of the Test Automation Pyramid*. Retrieved from mountaingoatsoftware.com: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley.

Gause, D., & Weinberg, G. (2011). *Exploring Requirements 1: Quality Before Design.* Dorset House Publishing Co Inc.

Goldratt, E. (1984). *The Goal.* North River Press.

Gottesdiener, E. (2005). *The Software Requirements Memory Jogger: A Pocket Guide to Help Software and Business Teams Develop and Manage Requirements.* Salem NH USA: GOAL/QPC.

IEEE 610.12. (1990). *Standard Glossary of Software Engineering.*

International Requirements Engineering Board e.V. (IREB). (2020). *Syllabus IREB Certified Professional for Requirements Engineering Foundation Level - Version 3.0.1.* Karlsruhe: IREB e.V.

ISO/IEC/IEEE 29148. (2011). *System and software engineering - Lilfe cycle processes - Requirements engineering.*

Jeffries, R. (1998, April 4). *You're NOT gonna need it!* Retrieved from ronjeffries.com: https://ronjeffries.com/xprog/articles/practices/pracnotneed/

Juran, J., & Blanton Godfrey, A. (1999). *Quality Control Handbook Fifth Edition.* New York: McGraw-Hill.

Larman, C., & Basili, V. (2003). Iterative and Incremental Development: A Brief History. *Computer*.

Larman, C., & Basili, V. (2003). Iterative and Incremental Development: A Brief History. *Computer*, 2.

Meyer, B. (1986). *Design by Contract.* Interactive Sofware Engineering Inc.

North, D. (2006). *Introducing BDD*. Retrieved from Dan North & Associates: https://dannorth.net/introducing-bdd/

North, D. (2009). Agile Specifications. *BDD and Testing eXchange.* London: Skillsmatter.

North, D. (2022, 3 15). *What's in a Story*. Retrieved from Dan North & Associates Ltd website: https://dannorth.net/whats-in-a-story/

Robertson, S., & Robertson, J. (2013). *Mastering the Requirements Process: Getting Requirements Right, 3 red.* Addison-Wesley.

Wiegers, K., & Beatty, J. (2013). *Software Requirements, 3 red.* Washington: Microsoft Press.

Wynne, M. (2015, 12 08). *Example Mapping introduction*. Retrieved from Cucumber.io: https://cucumber.io/blog/bdd/example-mapping-introduction/