

AMDARIS

REQUIREMENTS SPECIFICATION

October 2019 | Olesea Oaserele

© Amdaris Group Limited. All rights reserved.

WHAT WILL WE COVER

- Product Owner
- Requirements granularity and hierarchy
- Decomposing functional requirements
- User Stories
- Acceptance Criteria
- Splitting user stories
- Definition Of Ready (DoR)

PRODUCT OWNER

Product Owner is a **role in Scrum** team (we'll discuss about Agile frameworks during next lessons) responsible for maximizing the value of the product.

The Product Owner is the sole person responsible for **managing the Product Backlog**. Product Backlog management includes:

- Clearly expressing Product Backlog items.
- Ordering the items in the Product Backlog to best achieve goals and missions.
- Optimizing the value of the work the Development Team performs.
- Ensuring that the Product Backlog is visible, transparent, and clear to all, and shows what the Scrum Team will work on next.
- Ensuring the Development Team understands items in the Product Backlog to the level needed.

<https://www.scrum.org>



REQUIREMENTS GRANULARITY AND HIERARCHY

REQUIREMENTS GRANULARITY

- Requirements **granularity** is related to the level of details of each requirements.
- More details in the description of the expected behavior of the software – the finer the granularity.
- Different levels of granularity form the hierarchy of the requirements
- During the **requirements elicitation** and **documentation** process, the hierarchy and granularity can be established in different ways.
- Stakeholders typically tell you their wishes at various levels.
- PO cannot force the stakeholders to stick to a level of granularity
- PO has to create the hierarchy of requirements and granularity

CREATING LEVELS OF GRANULARITY

Establishing granularity and hierarchy.

- Top-down method – starting from visions and objectives to lower level requirements.
- Bottom-up method - grouping lower level requirements into larger chunks, ideally until re-creating the vision and objectives
- Middle-out method - starting with requirements in the middle, breaking some down into more detail while others are grouped together.

The most used method is middle-out, as the stakeholders rarely can deliver requirements in a structured way.

EXAMPLES

E-learning platform.

Coarse-grained requirements.

“As a student I want to learn about Agile in an online video course, so I do not have to go to a classroom“

“As a department head I want to be able to check the learning progress of all my students”

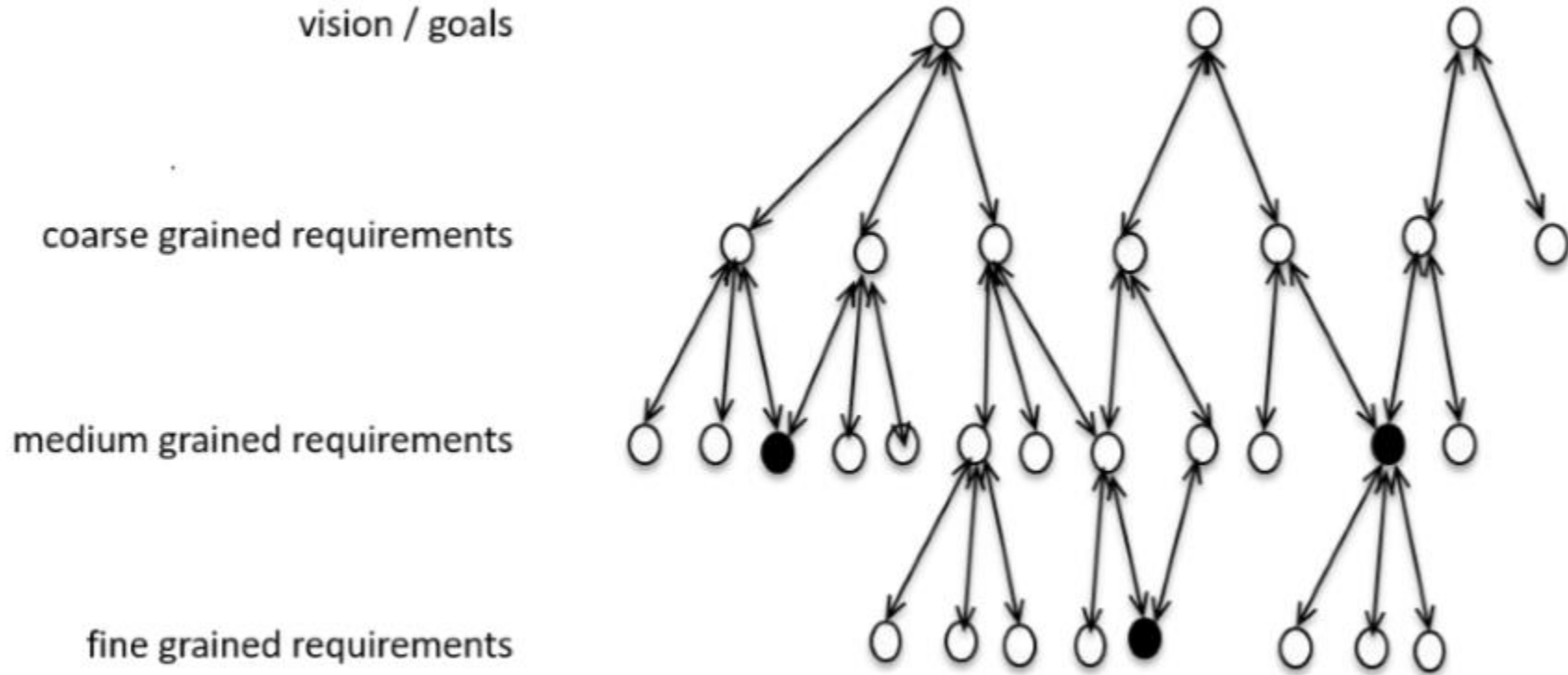
Fine-grained requirements.

“As a student while playing a video clip I want to be able to see the rest of its runtime in seconds”

“As a student while playing a video clip I want to be able to see the rest of its runtime in seconds colored in orange, blue, green, depending on % of time remaining. >50% - orange; 20 – 50% - blue; 0 – 20% - green”

Find the ambiguity in the last req.

REQUIREMENTS HIERARCHY



The **coarse-grained** requirements allow the product owner to keep an overview of all functional requirements necessary for:

- long term planning,
- road-mapping,
- selecting requirements that promise early business value for further investigation,
- high level estimation,....

The **fine-grained** requirements are necessary to achieve a thorough understanding of details that is necessary for the development team to implement those requirements.

There is no standard of how to represent requirements hierarchy.

In Agile the hierarchy **Theme – Epic – Feature - User Story – Task** is often preferred.


JUST IN TIME REQUIREMENTS (T-APPROACH)

Just in time (JIT) is a term coming from manufacturing and represents the fact that materials, goods, and labor are scheduled to arrive or be replenished exactly when needed in the production process.

In Agile the **just in time (JIT) requirements** technique is widely used.

- PO defines the coarse grained requirements first – **Decomposing the system**
- High level architecture is set first
- Requirements are refined gradually and... just in time
- PO selects for refinement and development the requirements which bring the highest value first – **Creating/splitting user stories**
- Architecture is revised along with detailed requirements
- It's very important to have timely conversations with stakeholders

Named **T-approach** because broad overview resembles the horizontal bar of the letter T while the drill-down of those requirements that promise highest business value resembles the vertical bar of the letter T.

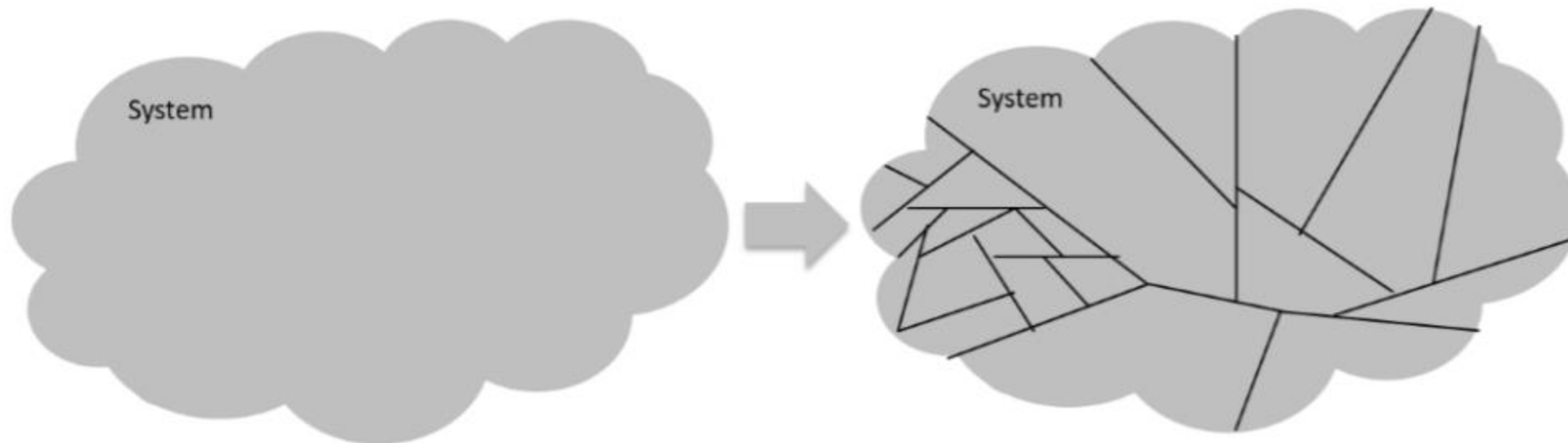


DECOMPOSING FUNCTIONAL REQUIREMENTS

Coarse graining

DIVIDE AND CONQUER

The **vision** and **business objectives** are not sufficient for developing a complex system. They have to be made **more precise** in order to come up with functional requirements that can be communicated to and implemented by the development team.



INVEST

The **INVEST** mnemonic is used for remembering the main attributes of a good quality Product Backlog Item (or User Story, or Functional Requirement) in Agile.

I – Independent. The PBI should be self-contained, in a way that there is no inherent dependency on another PBI.

N – Negotiable. PBIs are not explicit contracts and should leave space for discussion.

V – Valuable. A PBI must deliver value to the stakeholders.

E – Estimable. You must always be able to estimate the size of a PBI

S – Small. PBIs should not be so big as to become impossible to plan/task/prioritize within a level of accuracy.

T – Testable. The PBI or its related description must provide the necessary information to make test development possible.

DECOMPOSING APPROACHES (COARSE GRAINING)

- Split into **logical functions** (features, epics or themes)
Example for e-learning:
 - Establishing a contract (registering in the app)
 - Watching videos
 - Testing the knowledge
- Use **history** of the product to be developed or the **structure of an existing** similar product, as a partitioning theme. It will offer options for splitting the system in chunks of functionality based on earlier decompositions.

- Split by **organizational aspects** – meaning parts serving different departments or user groups

Example for e-learning:

- Functionality for students
- Functionality for admins
- Functionality for lecturers

- Split according to **hardware** or **environment**

Example for e-learning:

- App on desktop with responsive design
- App for mobile
 - iOS
 - Android

- Split by geographical distribution
Example for e-learning:
 - App for the larger number of potential users
 - Extension for different legislation in other regions
- Split by data (business objects)
Example for e-learning:
 - Functionality related to videos
 - Functionality related to text lectures
 - Functionality related to questions and quizzes
 - Functionality related to contracts and invoices

EXTERNALLY TRIGGERED, VALUE-CREATING PROCESS – SPECIAL TYPE OF DECOMPOSITION

- The only method which **starts with the context outside** of the system
- The **context diagram** is a valuable source when identifying external triggers, since it shows all adjacent systems that might request some action from the system under consideration.

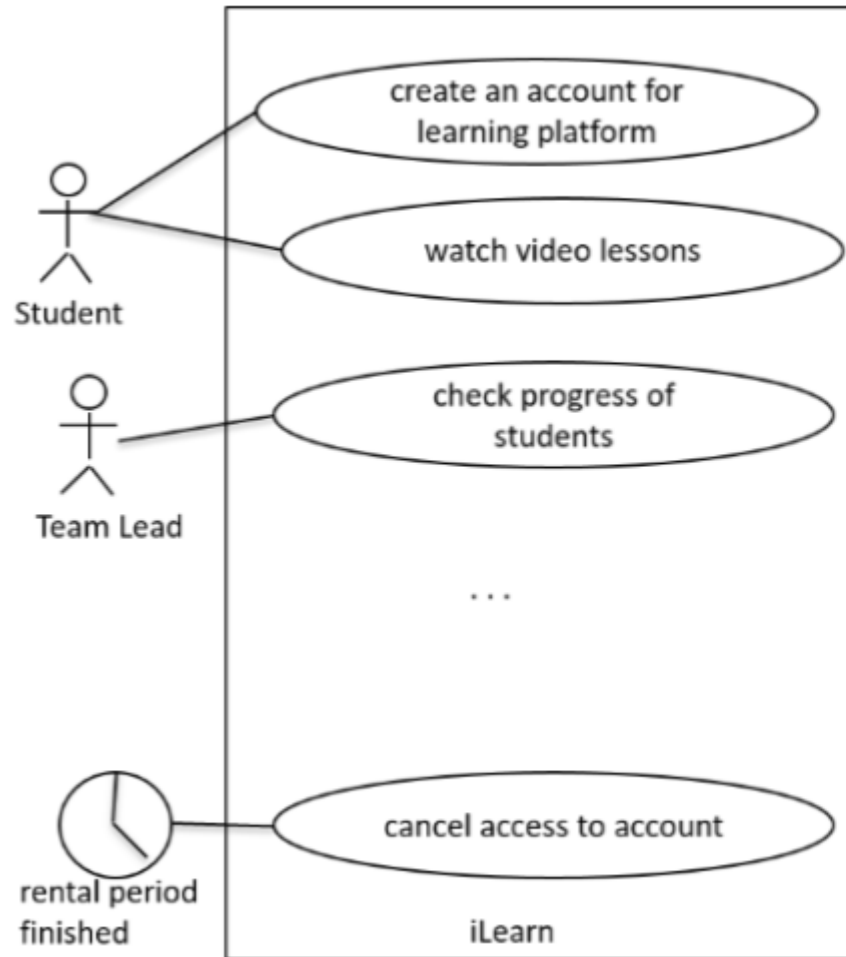
External triggers could have different sources:

- human users needing something from the system,
- other software systems sending input and requesting some system action,
- hardware devices (like sensors) triggering an action inside our system.

Has **different names**:

- Value-oriented decomposition
- Event-oriented decomposition
- Use case decomposition
- Business processes
- User Stories

EXAMPLE OF USE CASE DECOMPOSITION



As a student I want to create an account for the learning platform so that I can acquire RE knowledge everywhere

As a student I want to watch video lessons to prepare myself for the exam.

As a team lead I want to check the progress of my students

...

As owner of the portal I want to cancel access to learning accounts at the end of the rental period.

The resulting chunks of functionality **after coarse-grained decomposing** will respect the first 3 letters of the INVEST mnemonic.

I: independent of each other, meaning they are self-contained and minimize mutual dependencies. They should not overlap in concept, and we would like to be able to schedule and implement them in any order.

N: negotiable, meaning they do not yet represent a fixed contract, but leave space for discussions of the details.

V: valuable: they bring real value to the requester, that is to a person or another system in the context.

The EST part is more often obtained during fine-grained decomposing and user stories creation. Later in the course about this.

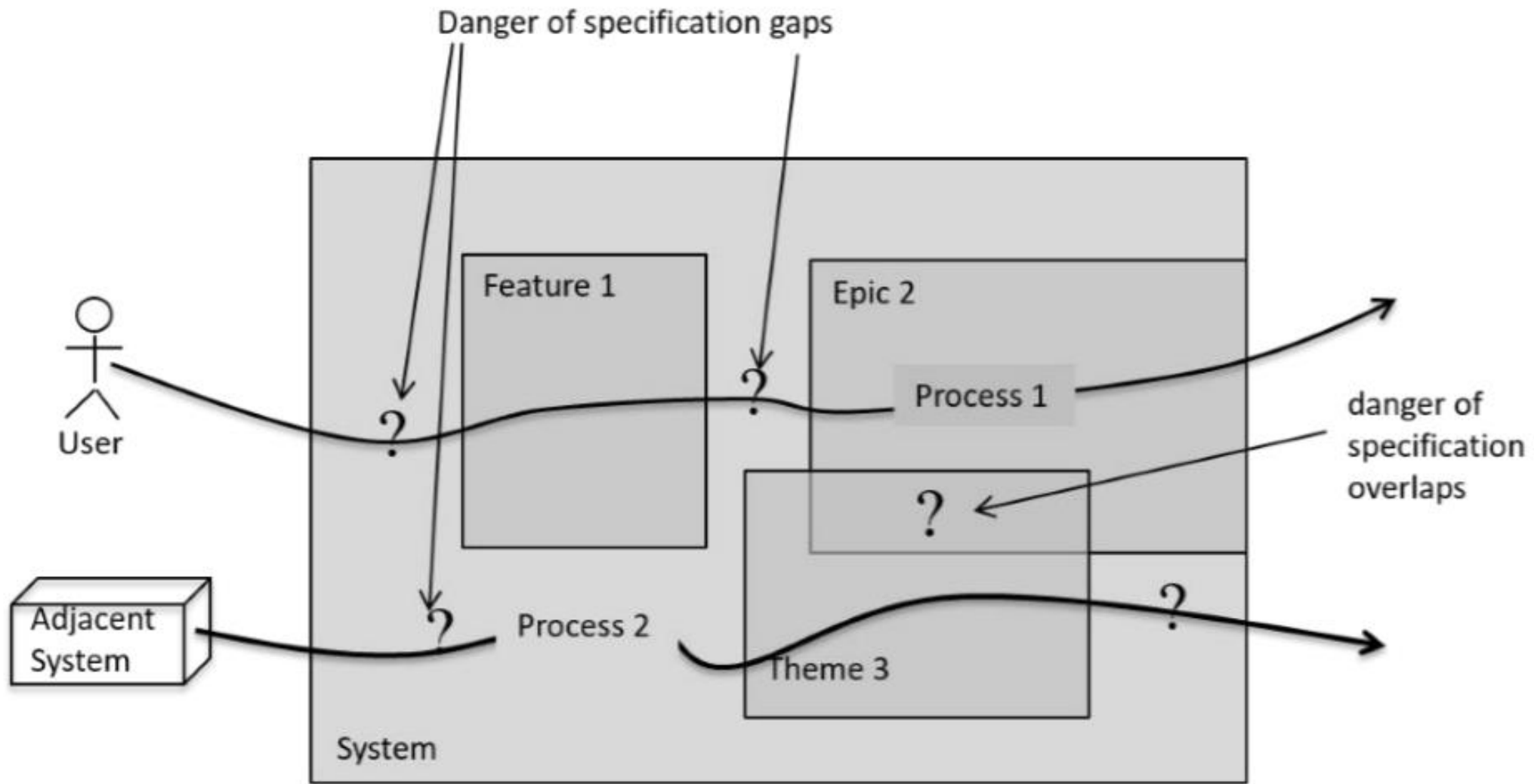
GAPS AND OVERLAPS DURING DECOMPOSITION

Regardless of the decomposition method used there is always the danger of obtaining gaps or overlaps in requirements specification. But thinking in terms of **value creating processes** avoids these dangers from the beginning.

A good practice for value-oriented process decomposition to identify events that happen in the context and to which the system has to react.

- **External** events: Triggered by users or adjacent systems;
Ex: “As a student I want to assess my knowledge with test questions.”
- **Temporal** events: Triggered by time or observation of system internal resources.
Ex: “Two weeks before the end of the subscription period it is time to remind students about a possible prolongation.”

NB A picture is worth 1000 words. Use the graphical representations as much as possible to ensure clear understanding and exclude ambiguity, gaps and overlaps.





USER STORIES

Fine graining

WHAT ARE USER STORIES

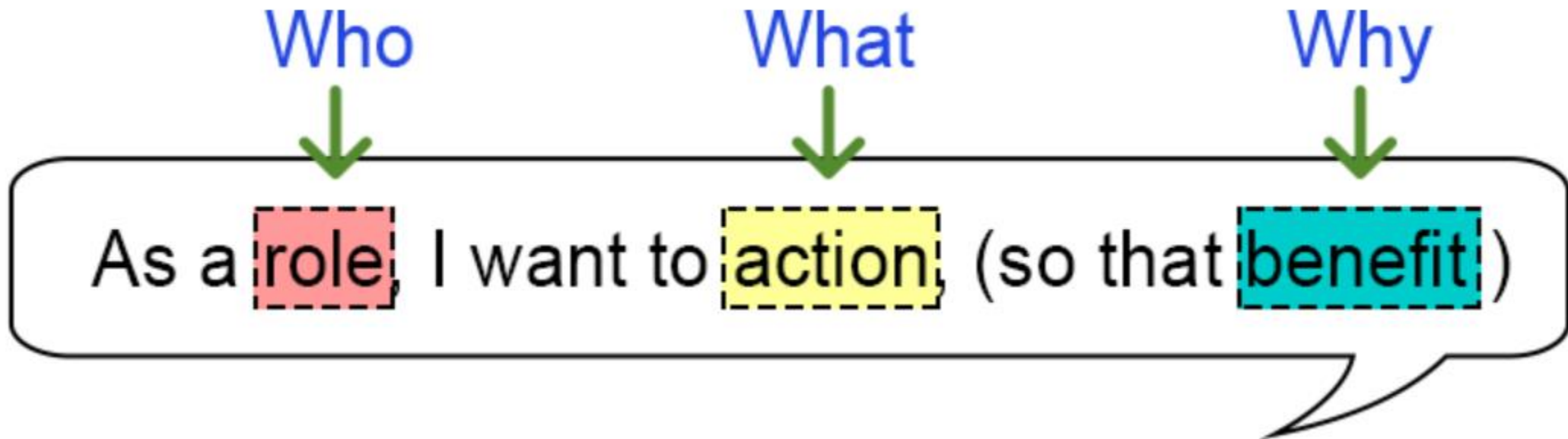
User stories are a popular approach to structure requirements. It captures the who, what and why of a functionality. The three constituents of a US ensure that:

1. we have someone **who** wants that functionality (“As a user ...”),
 2. we know **what** the user wants (“... I want ...”) and
 3. we understand the **reason** or motivation (“... so that ...”).
- It is not so much the formalism that makes user stories successful, as it’s asking and answering these three questions.
 - Note that sometimes the word “user” is a bit misleading, since the person wanting a feature is not necessarily the one working with the system as a user

USER STORIES TEMPLATE

The most widely used template of USs is:

- **Who** it is for?
- **What** it expects from the system?
- **Why** it is important?



Role - The user should be an actual human who interacts with the system.

- Be as specific as possible
- The development team is NOT a user

Action - The behavior of the system should be written as an action.

- Usually unique for each User Story
- The "system" is implied and does not get written in the story
- Active voice, not passive voice ("I can be notified")

Benefits - The benefit should be a real-world result that is non-functional or external to the system.

- Many stories may share the same benefit statement.
- The benefit may be for other users or customers, not just for the user in the story.
- The benefits of many stories will create a business objective

EXAMPLES

As a [customer], I want [shopping cart feature] so that [I can easily purchase items online].

As a [manager], I want to [generate a report] so that [I can understand which departments need more resources].

As a [customer], I want to [receive an SMS when the item is arrived] so that [I can go pick it up right away]

Role represents the person, system, subsystem or any entity else who will interact with the system to be implemented to achieve a goal. He or she will gain values by interacting with the system.

Action represents a user's expectation that can be accomplished through interacting with the system.

Benefits represents the value behind the interaction with the system.

THE 3 C MODEL

User Stories are often written on physical or virtual cards and arranged on physical or virtual walls to facilitate planning and discussions.

This aspect was summarised in the 3C-model (**Card, Conversation, Confirmation**) to distinguish the more social character of stories from the more documentary character of other requirement notations.

Card

Written on card.

Conversation

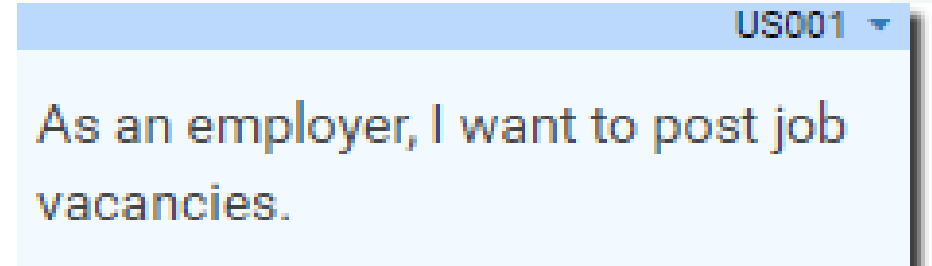
Details captured in
conversations.

Confirmation

Acceptance criteria confirm
that the story is done.

CARD

- Card represents 2-3 sentences used to describe the intent of the story that can be considered as an **invitation to conversation**.
- The card serves as a **memorable token**, which **summarizes intent** and represents a more detailed requirement, whose details remain to be determined.
- You don't have to have all of the Product Backlog Items written out perfectly "up front", before you bring them to the team. It acknowledges that **the customer and the team will be discovering** the underlying business/system needed as they are working on it.
- This **discovery occurs through conversation** and collaboration around user stories.
- The Card is usually follows the format similar to : As a (**role**) of the product, I can (do **action**) so that I can obtain (some **benefits** / value)



CONVERSATION

Conversation represents a **discussion** between the target users, team, product owner, and other stakeholders, which is necessary **to determine the more detailed behavior** required to implement the intent. In other words, the card also represents a "promise for a conversation" about the intent.

- The collaborative conversation facilitated by the Product Owner which involves **all stakeholders and the team**.
- The conversation is where the **real value of the story** lies and the written Card should be adjusted to reflect the current shared understanding of this conversation.
- This conversation is **mostly verbal** but most often supported by documentation and ideally automated tests of various sorts (e.g. Acceptance Tests).



Conversation Confirmation Scenario Storyboard Design Reference History

Story Map

Estimate & Spike

Sprint

Configuration

B I A ^ v ☰ ☷

- Agreement of statement of Purposes for Personal Data required
- P1 - Employment information
 - General employer vs Employment agency
 - Business reg certificate no.
 - Sector/industry
 - Company name
 - Contract details (name, tel, fax, email address)
- P2 - Vacancy information
 - Job title
 - Job duties
 - Contract details - perm? temp? summer job?
 - Working hrs
 - Salary - basic salary, double pay, other benefits
 - Edu level
 - Working exp?



CONFIRMATION

Confirmation represents the **Acceptance Tests**, which is how the customer or product owner will **confirm that the story has been implemented** to their satisfaction. In other words, Confirmation represents the **conditions of satisfaction** that will be applied to determine whether or not the story fulfills the intent as well as the more detailed requirements.

- The Product Owner must confirm that the story **is complete** before it can be considered "done"
- The team and the Product Owner check the "doneness" of each story in light of the Team's current **definition of "done"**
- Specific acceptance criteria that is different from the current definition of "done" can be established for individual stories, but the current **criteria** must be well understood and **agreed to by the Team**. All associated acceptance tests should be in a passing state.

JobsDIR - Visual Paradigm Enterprise

Dash Project ITSM UeXceler Diagram View Team Tools Modeling Window Help

Conversation Confirmation (0/10) Scenario Storyboard Design Reference History

B I A ^ v

<input checked="" type="checkbox"/>	A summary of vacancy is displayed by submitting the form with all compulsory fields filled.	
	1. Click on Post Vacancy	Pre-Vacancy Order Form presented
	2. Select General Employer for Employer Type	
	3. Click Next	Vacancy Order Form presented
	4. Fill in all fields marked *	
	5. Click Next	Summary of vacancy presented
<input type="checkbox"/>	A summary of vacancy is displayed by submitting the form with all fields filled.	
<input type="checkbox"/>	The field 'General employer vs Employment agency' will be focused by submitting the form with this field unspecified.	
<input type="checkbox"/>	The field 'Business registration certificate no.' will be focused by submitting the form with this field unspecified.	
<input type="checkbox"/>	The field 'Sector/Industry' will be focused by submitting the form with this field unspecified.	
<input type="checkbox"/>	The field 'Company name' will be focused by submitting the form with this field unspecified.	
<input type="checkbox"/>	The field 'Contact details' will be focused by submitting the form with this field unspecified.	
<input type="checkbox"/>	The field 'Job title' will be focused by submitting the form with this field unspecified.	
<input type="checkbox"/>	The field 'Job duties' will be focused by submitting the form with this field unspecified.	
<input type="checkbox"/>	The field 'Contract details' will be focused by submitting the form with this field unspecified.	

Story Map
Estimate & Spike
Sprint
Configuration

USER STORIES SHOULD BE INVEST

User Stories should respect the INVEST mnemonic but...

- It's almost impossible to have all 3 letters applied
- The trade-offs should be thought of
- Usually INV are least accurate for user stories and more accurate for coarse-grained requirements (features, epics)
- Usually EST are implemented accurately for user stories and not taken in account so much for coarse-grained requirements

BENEFITS OF USER STORIES

Benefits for adopting user story approach in agile development:

- The simple and consistent format **saves time** when capturing and prioritizing requirements while remaining **versatile** enough to be used on large and small features alike.
- Keep yourself **expressing business value** by delivering a product that the client really needs
- **Avoid** introducing **detail too early** that would prevent design options and inappropriately lock developers into one solution.
- **Avoid** the appearance **of false completeness** and clarity
- Get to small enough chunks that **invite negotiation** and movement in the backlog
- Leave the technical functions to the architect, developers, testers,



ACCEPTANCE CRITERIA

For any types of granularity

For any template

ACCEPTANCE CRITERIA MAIN PURPOSES

Acceptance criteria (AC) are the **conditions** that a software product must **meet to be accepted** by a user, a customer, or other system. AC are written **from the end-user's perspective** and ensure that all stakeholders and users are satisfied with what they get.

- **Feature scope detalisation.** AC define the **boundaries** of user stories. They provide **precise details** on functionality that help the team understand whether the story is completed and works as expected.
- **Describing negative scenarios.** AC define scenarios of wrong usage or alternative flows and explain how the system must react on them.
Ex: Requirement that the system to recognize unsafe password inputs and prevent a user from proceeding further. Invalid password format is an example of a so-called negative scenario when a user does invalid inputs or behaves unexpectedly.

- **Setting communication.** Acceptance criteria synchronize the visions of the client and the development team. They ensure that everyone has a common understanding of the requirements: Developers know exactly what kind of behavior the feature must demonstrate, while stakeholders and the client understand what's expected from the feature.
- **Streamlining acceptance testing.** AC are the **basis** of the user story **acceptance testing**. Each acceptance criterion must be **independently testable** and thus have a clear pass or fail scenarios. They can also be used to verify the story via automated tests.
- **Feature estimation.** Acceptance criteria specify what exactly must be developed by the team. Once the team has precise requirements, they can split user stories into **tasks that can be correctly estimated**.

AC TYPES AND STRUCTURE

Acceptance criteria (AC) can be written differently

- scenario-oriented (Given/When/Then) – used for BDD
- rule-oriented (checklist)
- custom formats – invent your own format

NB: Even natural informal language can be used, it is NOT recommended, as it is prone to errors and ambiguity

SCENARIO-ORIENTED FORMAT

Scenario-oriented format of writing AC is known as

- The *Given/When/Then* (GWT) type
- Gherkin language
- Behavior Driven Development format

Given some precondition

When I do some action

Then I expect some result

This format provides a consistent structure that helps testers define **when to begin and end testing** a particular feature. It also reduces the time spent on writing test cases as the behavior of the system is described upfront.

Each acceptance criteria written in this format has the following statements:

- **Scenario** – the name for the behavior that will be described
- **Given** – the beginning state of the scenario
- **When** – specific action that the user makes
- **Then** – the outcome of the action in “When”
- **And** – used to continue any of three previous statements

When combined these statements cover all actions that a user takes to complete a task and experience the outcome

Example:

User story: As a user, I want to be able to recover the password to my account, so that I will be able to access my account in case I forgot the password.

Scenario1: Forgot password – send link

Given: The user has navigated to the login page

When: The user selected *forgot password* option

And: Entered a valid email to receive a link for password recovery

Then: The system sent the link to the entered email

Scenario2: Forgot password – navigate through the received link

Given: The user received the link via the email

When: The user navigated through the link received in the email

Then: The system enables the user to set a new password

RULE-ORIENTED FORMAT

In some cases, it's difficult to fit acceptance criteria into the Given/When/Then structure. For instance the following cases:

- You're working with user stories that describe the system level functionality that needs other methods of quality assurance.
- The target audience for acceptance criteria doesn't need precise details of the test scenarios.
- GWT scenarios don't fit to describing design and user experience constraints of a feature. Developers may miss a number of critical details.

Rule-oriented format can be used in these cases. It represents a set of rules that describe the behavior of a system. Based on these rules, you can draw specific scenarios.

Example

User story: *As a user, I want to use a search field to type a city, name, or street, so that I could find matching hotel options.*

Basic search interface acceptance criteria

- The search field is placed on the top bar
- Search starts once the user clicks “Search”
- The field contains a placeholder with a grey-colored text: “Where are you going?”
- The placeholder disappears once the user starts typing
- Search is performed if a user types in a city, hotel name, street, or all combined
- Search is in English, French, German, and Ukrainian
- The user can’t type more than 200 symbols
- The search doesn’t support special symbols (characters). If the user has typed a special symbol, show the warning message: “Search input cannot contain special symbols.”

BEST PRACTICES OF WRITING AC

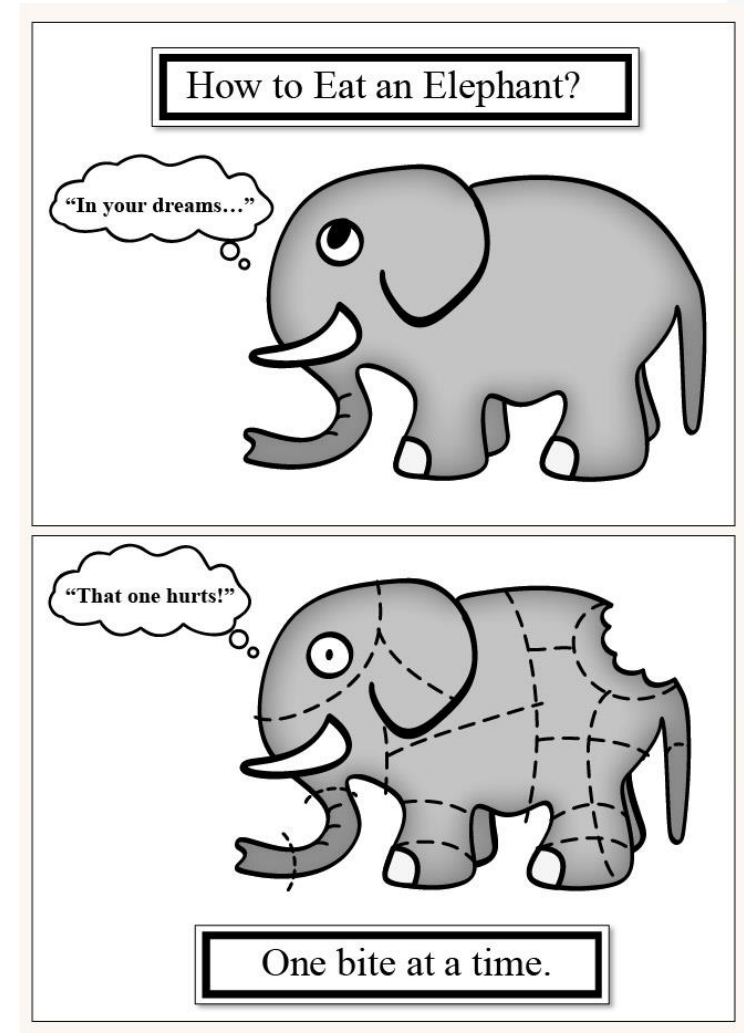
- **Document criteria before development.** This way, the team will likely capture all customer needs in advance.
- **Don't make AC too narrow.** AC must convey the intent but not a final solution. AC should leave developers enough space for creating the best possible solution.
- **Keep your criteria achievable.** Effective acceptance criteria define the reasonable minimum chunk of functionality that you're able to deliver. Too many details can block the team or bring into analysis paralysis.
- **Keep AC measurable and not too broad.** Broad acceptance criteria make a user story vague.
- **Avoid technical details.** Stakeholders or managers may not have technical background. Leave technical details to the dev team
- **Reach consensus.** Make sure that you've communicated your AC to stakeholders and reached a mutual agreement.
- **Write testable AC.** This will allow testers to verify that all requirements were met.



SPLITTING USER STORIES

WHY TO SPLIT REQUIREMENTS IN USER STORIES

- To be able to eat the elephant.
- To focus on the most important 80% according to Pareto. https://en.wikipedia.org/wiki/Pareto_principle
- To be able to trim the tail when needed
- To get conform to INVEST
- Clarity of value added in a time frame
- Clarity for development team
- More exact estimations
- Less probability of Gold Plating
- Earlier testing to fix
- Earlier feed-back for correcting the course



3 FIRST STEPS AND A WIDE SPREAD ADVICE

1. Distillate what's important and high priority
Deprioritize or throw away low priority and low value.
2. Slice vertically – User Goal decomposition
Every US should bring a visible value
3. Chop what's important to small pieces.
Get more equally sized small stories

A wide-spread advice:

Each US should be $1/10$ – $1/6$ from current velocity.

Questionable?

SPLITTING TECHNIQUES

1. Split by workflow and alternative flows
 - Each happy path with its User Story
 - Alternative flows with their User Stories
 - Each actor actions with their User Stories
 - Focus on happy paths first
2. Split summary Goal into User Goals
 - 'I Want' of the summary goal = sum of 'So That' of the user goals
 - Split by actor's actions, UI elements, validations, etc.
 - Focus on the most important part first not on sequence
3. Split into steps
 - Sometimes dependency is needed. The I is lost.
 - Make sure of splitting vertically
 - Create a skeleton, then fill in the gaps

4. Split by user groups

- User roles
- User groups by region
- User groups by age
- Individuals vs Organisations

5. Split by business rule variations

- Identify all variations of business rules
- Group similar variations in one User Story, if possible
- Focus on User Stories covering more rules

6. Split by major effort

- Split the most effort in first US
- Focus on the major effort first
- The other parts could be deferred if needed

7. Simple/Complex

- Bronze-Plated Flows vs Gold-Plated Flows
- First make it work, then add details
- Capture the simplest version of the functionality in a US
- Break out variations and complexities in other stories
- Focus on the basic US first

8. Split by data variation

- Data variations = groups of data needed to describe an entity
- Split US for each data variation or for groups
- Focus on the most simple group of data first
- Describe details in subsequent User Stories

9. Data entry methods

- Create US for entering data the simplest way
- Add constraints in separate US
- Add data-entry functionalities in separate US

10. Defer Performance -Split out NFRs

- First make it work
- Then make it fast
- Add NFRs in separate stories

11. Operations

- Separate User Story for each CRUD operation
- US for Replicate and Duplicate

999. Split out a spike

- When there is uncertainty, there is a spike
- Spike is to be created only when nothing else helps
- Spike = unclear objective and timescales
- Risk of getting carried away in investigation
- Clear User Stories should be the result of spikes

GOOD PRACTICES

- Keep the team focused on high value by splitting out the low value and high effort bits
- Split them in many rounds
- Stop when the size of US is comfortable for the team
- Do analysis first – there's no good splitting without understanding
- Re-prioritise regularly so that the low values go down
- Split vertically



DEFINITION OF READY

WHY IS DEFINITION OF READY IMPORTANT

- To ensure the US is clearly understood by the team
- To avoid reworks and waste of PO and team's time
- To avoid bugs and large refactoring
- The DoR should be agreed with stakeholders and the team
- Having the well structured DoR will bring benefits:
 - Measure a backlog item's "ready" state
 - Help the team identify when the product owner or another team member becomes overwhelmed
 - Keep the team accountable to each other
 - Reduce pressure on the team to commit to estimates before stories are "Ready"
 - Reduce "requirements churn" in development

EXAMPLE

Definition of Ready for a User Story

- User Story defined
- User Story dependencies identified
- User Story sized by Delivery Team
- Scrum Team accepts User Experience artefacts
- Performance criteria identified, where appropriate
- Person who will accept the User Story is identified
- Team has a good idea what it will mean to Demo the User Story

THANK YOU

AMDARIS

© Amdaris Group Limited. All rights reserved.