University of Southampton

Modelling Requirements

COMP6226: Software Modelling Tools and Techniques for Critical Systems

Dr A. Rezazadeh (Reza)
Email: ra3@ecs.soton.ac.uk or ar4k06@soton.ac.uk

October 24

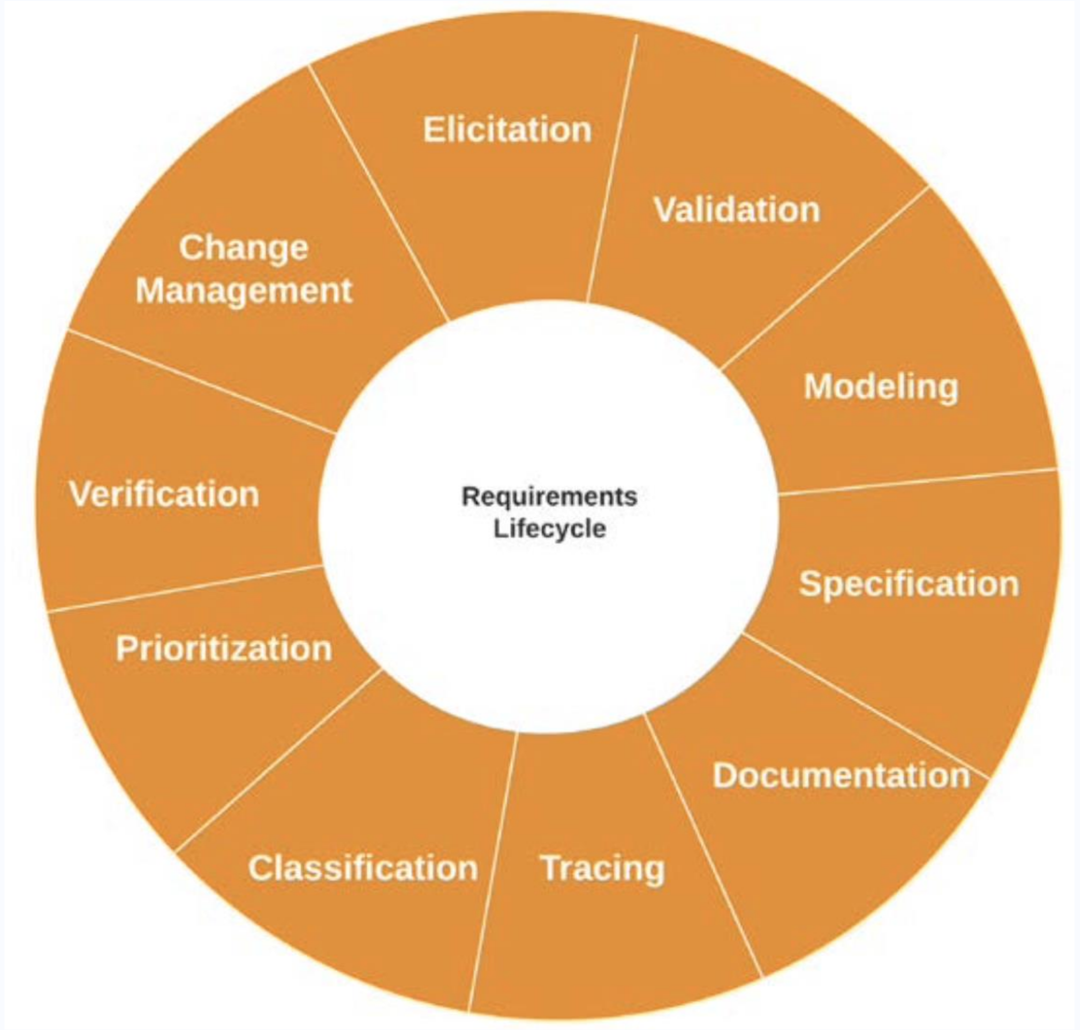
Overview

- In this Lecture, we will cover the following topics:
  - Modelling requirements with impact maps
  - Identifying capabilities and features
  - Knowing the difference between functional and non-functional requirements
  - Introducing behaviour-driven development

Objectives

- By the end of this lecture:
  - – you will know how to distinguish *capabilities* from *features*
  - – and *model basic requirements* with an *impact map*
  - – and *understand how BDD fits in with impact mapping*
  - – and how we *can model all types of requirements* using the techniques presented in this lecture.

Requirements life cycle stages

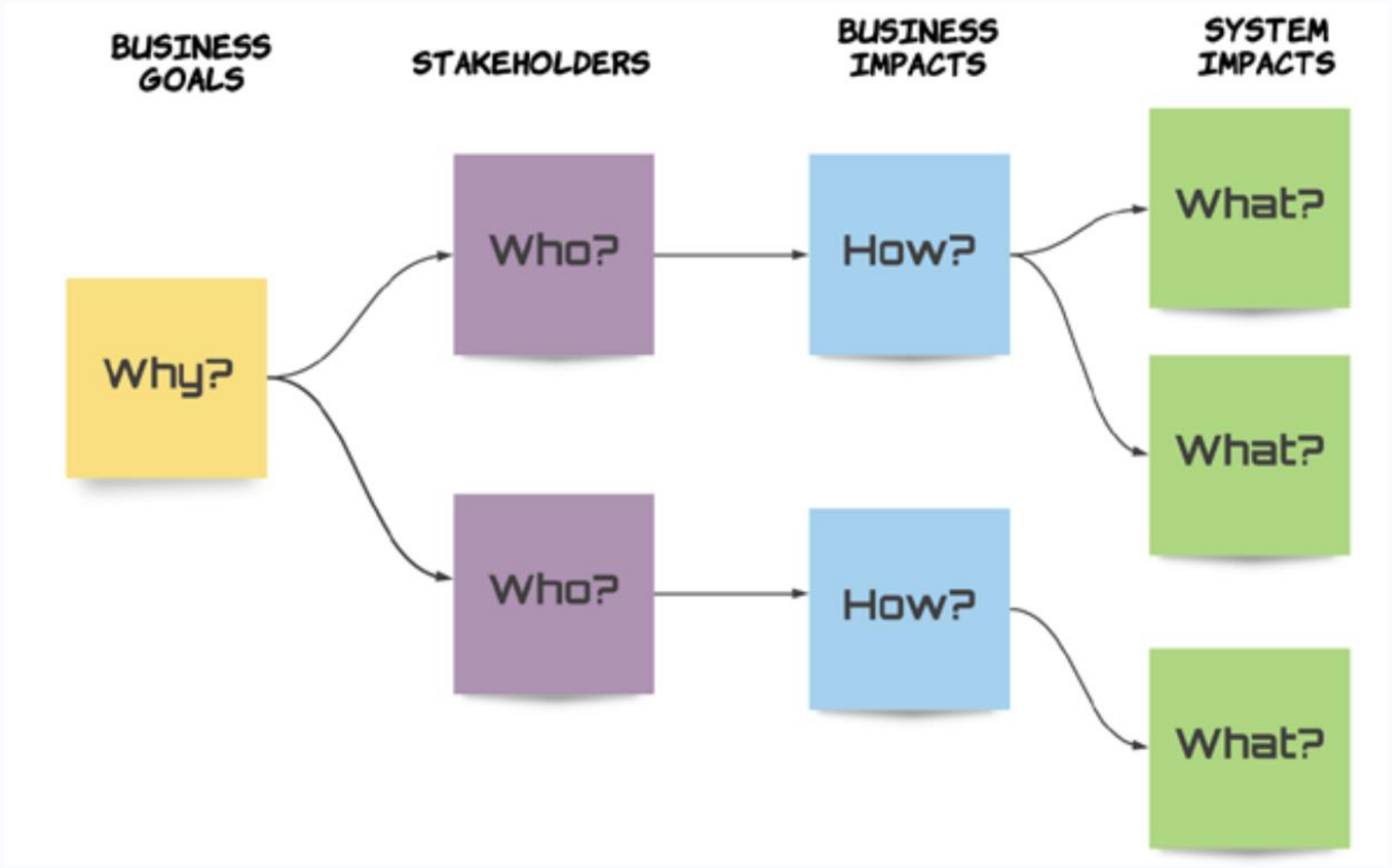


Impact mapping

- *Impact map* is a tree graph with four levels, where each level of the tree represents an answer to some fundamental questions about our system:
  - *Why are we building the system?*
  - *Who benefits from it?*
  - *How can the stakeholders achieve their goals?*
  - *What can the system do to help the stakeholders achieve their goals?*

- *Impact mapping* is a technique that he evolved from UX-based effect-mapping methods in order to improve communication, collaboration, and interaction in teams and organisations.

Impact Mapping – Graphical Representation



Impact Mapping – Interpretation

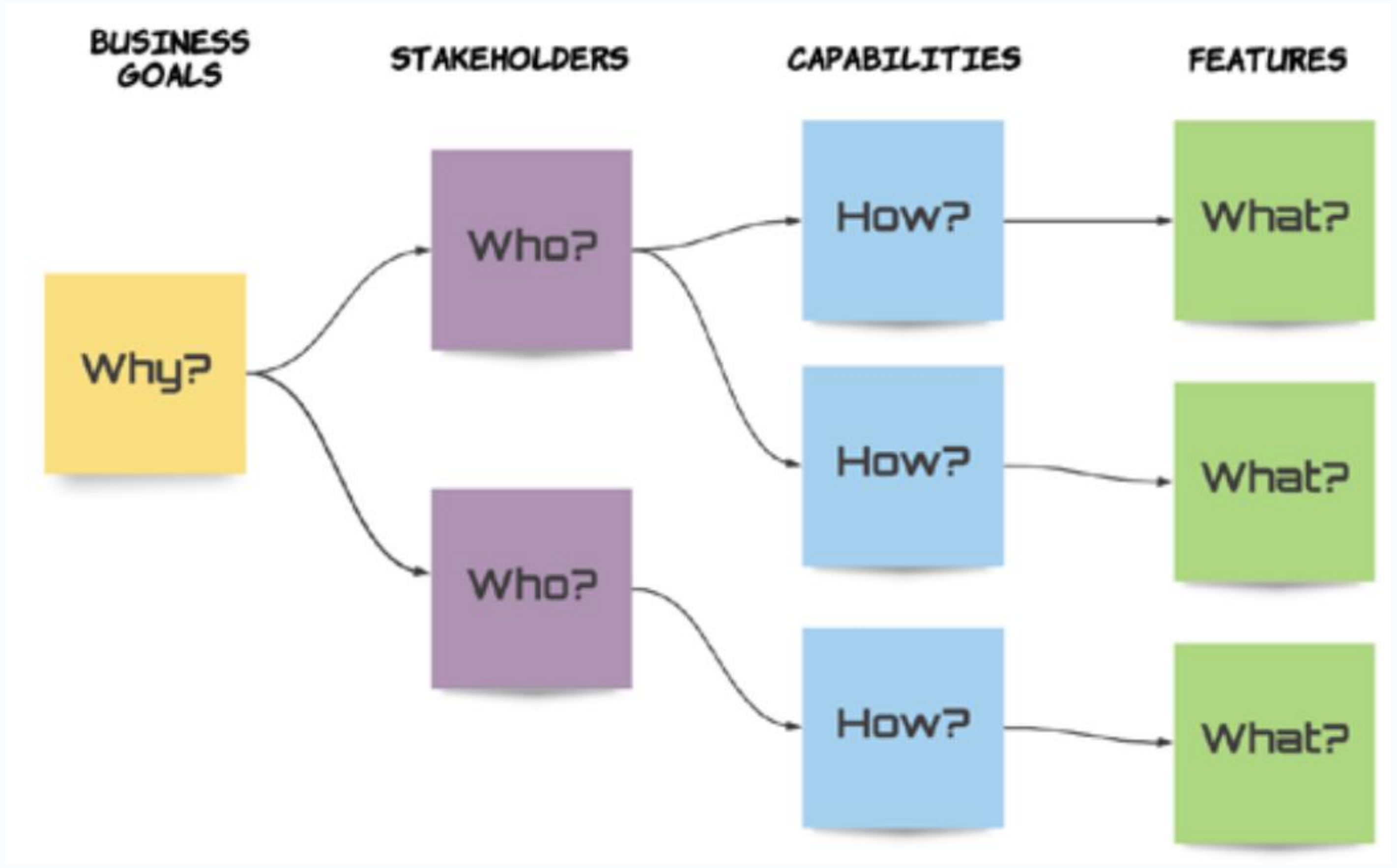
- We start by identifying a *business goal* (the why) and the stakeholders who are striving for it or are affected by it (the who).
- We then ask, "*how should our stakeholders' behaviour change so that the goal is accomplished?*"
  - This is the business impact of the stakeholders trying to realize their goal.
- The next question is "*what can we do, as an organisation or a team, to support the required impacts?*"
  - These are the things that our organization needs to deliver to support the stakeholders' behaviour.
  - From the perspective of a software team, these are the *system impacts*. They are our system's features – that is, the functionality we will need to provide to the stakeholders to help them realise the business or domain impacts they need to have.

Identifying capabilities and features

- In previous lecture, we identified two of the main entities in the requirements domain: stakeholders and goals.
- In the previous slides about impact mapping, we saw how these entities slot perfectly into an impact map.
- Earlier we also saw how the third and fourth levels of an impact map correspond to the business and system impacts of a stakeholder's effort to accomplish their goal.
- We define the business impact as a ***capability***.
  - A capability is a stakeholder's required ability to do something with our system in order to reach their goal.
- We define the system impact as a ***feature***.
  - A feature is a system functionality or behaviour required in order to support a capability.

Impact map with the new definitions

- **Goal (why)**: The intended benefit of our system
- **Stakeholder (who)**: Someone who interacts with, benefits from, or is otherwise
- affected by our system
- **Capability (how)**: A system ability that enables stakeholders to achieve a goal
- **Feature (what)**: A functionality that helps support a capability



The key difference between capabilities and features

- The key differentiation between **capabilities** and **features** is that the capabilities are discovered by answering the question "*how can the stakeholder accomplish their goal?*", while features are discovered by answering the question "what functionality must the system provide in order to deliver these capabilities to the stakeholders?"

- Note: Some people use the terms *high-level features* and *low-level features*, or *features, epics, and user stories*, to denote the meaning of capabilities and *features*.

What is a capability?

- A ***capability*** is a system ability that enables a stakeholder to achieve a goal.
  - It encapsulates the impact that the stakeholder has on our system in order to successfully realise their goal.
  - Capabilities reflect domain- or business-level requirements that don't describe or prescribe a particular implementation.
- Here are some capability examples:
  - Example 1: Sarah is a seller on a marketplace app. Her goal is to ***sell her items as quickly as possible***. To achieve her goal, she needs the *capability* to make her stock more visible to buyers.
  - Example 2: Tom is a seller on a marketplace app. He wants to empty his house of bulky items he no longer needs. To achieve this goal, he wants the *capability* to offer buyers a discount if they can pick up the item from his house.

Capabilities – Another example

- Example 3: Jane is the CTO of the company that builds the marketplace app. One of her goals is to increase the number of people using the app. To achieve her goal, she needs the capability to *offer new visitors to the website incentives to make them register as members*.

- *Capabilities* define how our stakeholders will impact our system's behaviour, but not what the system behaviour will be.
- Our job as system builders is to support and deliver these capabilities. We do this by implementing features.

University of Southampton

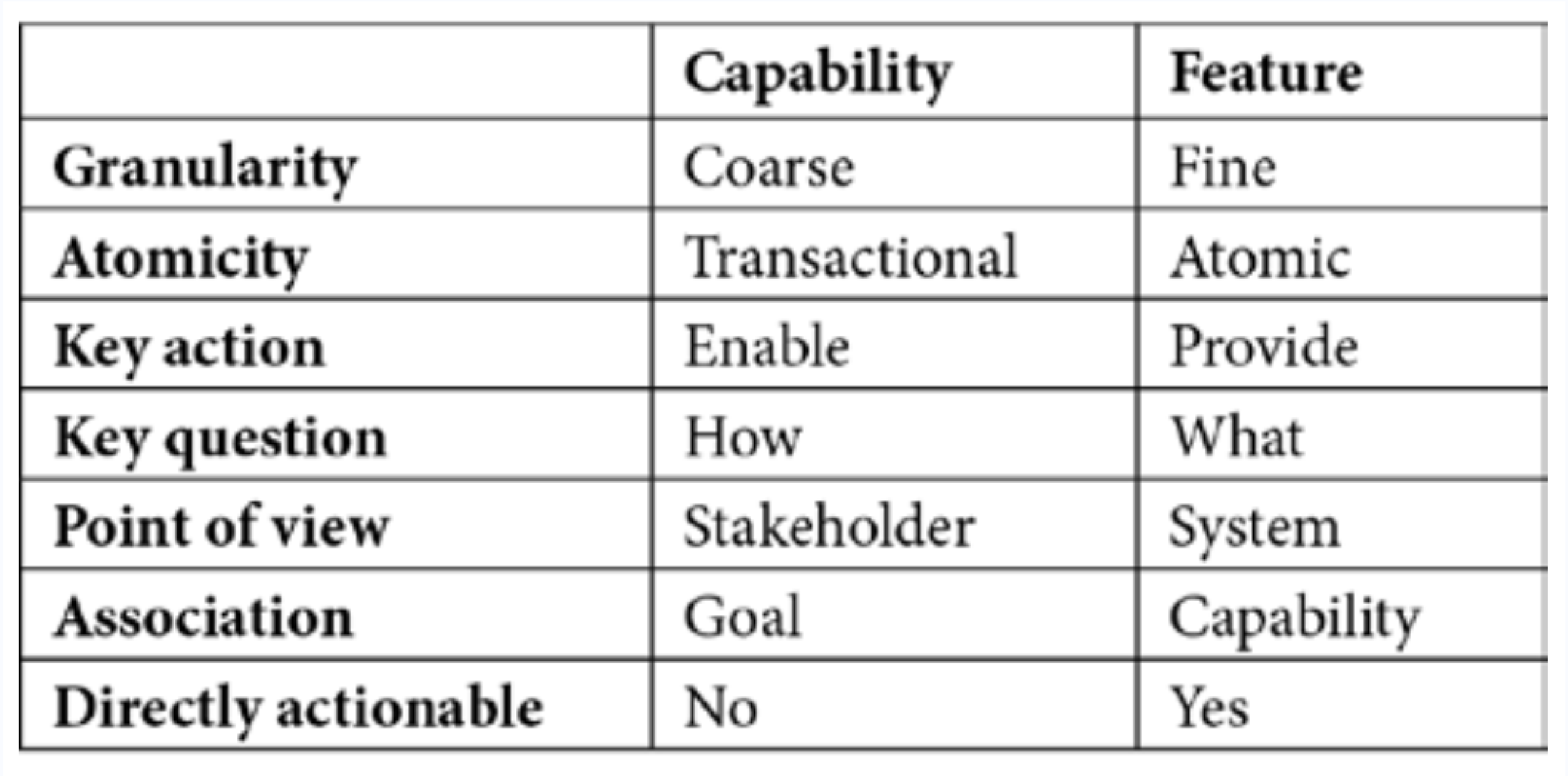
What is a feature?

- A **feature** describes a *system behaviour*.
  - It's closely associated with a *capability* and answers the question "*what system functionality do we need to implement in order to deliver this capability?*"
  - If a *capability* represents the *impact the stakeholder has on our system*, then a *feature* represents the *impact that capability has on the development team*.
  - *Features* describe *the system behaviour*, not its *design* or *architecture*.
  - In other words, they *describe what the system does*, from *an actor's perspective*.
  - Features are usually structured in the form of a *title*, *some descriptive information* or a *user story*, and *a number of acceptance criteria* (we'll be referring to those as *scenarios*).

Some examples of features

- *Example 1*: Sarah is a seller on a marketplace app, who needs the *capability* to *make her stock more visible to buyers*.
  - Some *features* that could help support this capability are *placing Sarah's stock on top of the stock listings and sending marketing emails about Sarah's stock to prospective buyers*.
- *Example 2*: Tom is a seller in a marketplace app, who wants the *capability* to *offer buyers a discount if they can pick up the item from his house*.
  - A *feature* that would support this capability is *adding money-off stickers to Tom's items for sale, promoting this discount*.
- *Example 3*: Jane is the CTO of the company that builds the marketplace app. She requires the *capability* to *offer incentives to new visitors to the website in order to make them register as members*.
  - *Displaying the membership benefits*, such as *free delivery*, *at the users' checkou*t is a *feature* that would help realise this capability.

Criteria to distinguish between capabilities and features



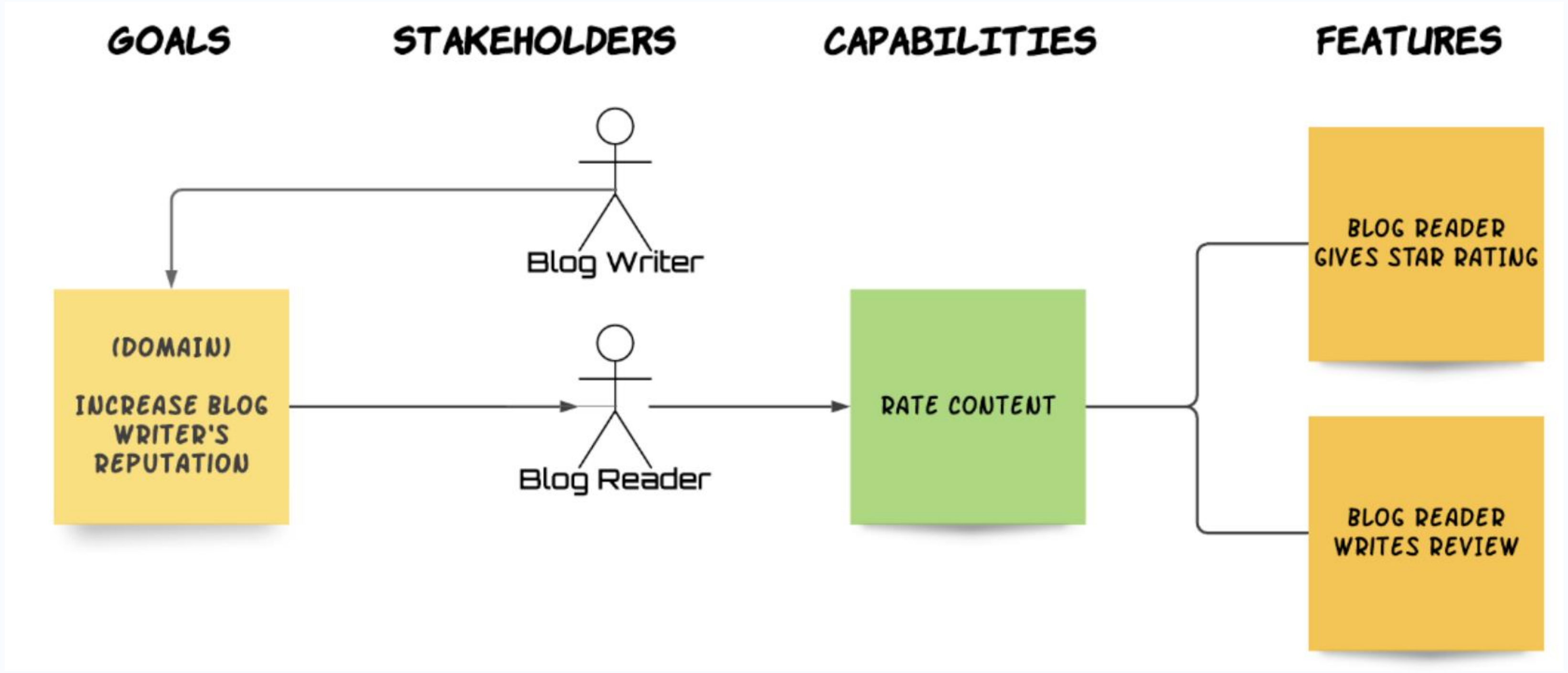
	Capability	Feature
Granularity	Coarse	Fine
Atomicity	Transactional	Atomic
Key action	Enable	Provide
Key question	How	What
Point of view	Stakeholder	System
Association	Goal	Capability
Directly actionable	No	Yes

Use case 1 – Content rating requirements for a knowledge-sharing platform

- We are building a knowledge-sharing platform where software developers can exchange tips, advice, and other forms of knowledge.
  - Many developers posting blog posts on our platform tell us that they would like to have their posts rated by other developers so that they can build up a reputation as knowledgeable and effective developers.
  - Here, the domain goal is to increase posters' reputations.
  - to accomplish this goal, blog posters need to be able to have their content rated by blog readers.
  - This is a capability that can identified as it directly relates to a goal.
  - The next step is to try to derive some features that will help us deliver this capability.
  - To do that, we – the system builders – need to communicate with both the posters and readers about the best ways we can support this capability. After doing so, we produce two pieces of functionality.

Producing functionality

1. Use a star-rating system, where the reader gives a post a rating of 1 to 5 stars, depending on how useful they thought the post was.
2. Enter textual feedback, where the reader can add some comments about the things they liked or didn't like in the post.
   - These two ways of delivering the capability are our features:


GOALS
STAKEHOLDERS
CAPABILITIES
FEATURES
Blog Writer
(DOMAIN) INCREASE BLOG WRITER'S REPUTATION
Blog Reader
RATE CONTENT
BLOG READER GIVES STAR RATING
BLOG READER WRITES REVIEW

Some notes about the previous diagram

- Here, an actor's goal is being realized through another actor's impact.
  - To help the blog writer reach their goal, we need to provide a blog reader with the capability to rate the blog writer's content.
- The main actor involved in this capability – that is, the blog reader – does not have an inherent goal to realise by leveraging the capability.
  - They are in fact helping another actor realise their own goal.
  - This is absolutely fine. In fact, cases like these help us explore the domain more and discover new goals and capabilities.
- So, in this case, we could ask ourselves "*how can we encourage blog readers to write more reviews?*"
  - We could easily discover that we need additional capabilities, such as awarding badges for readers who write the most reviews.

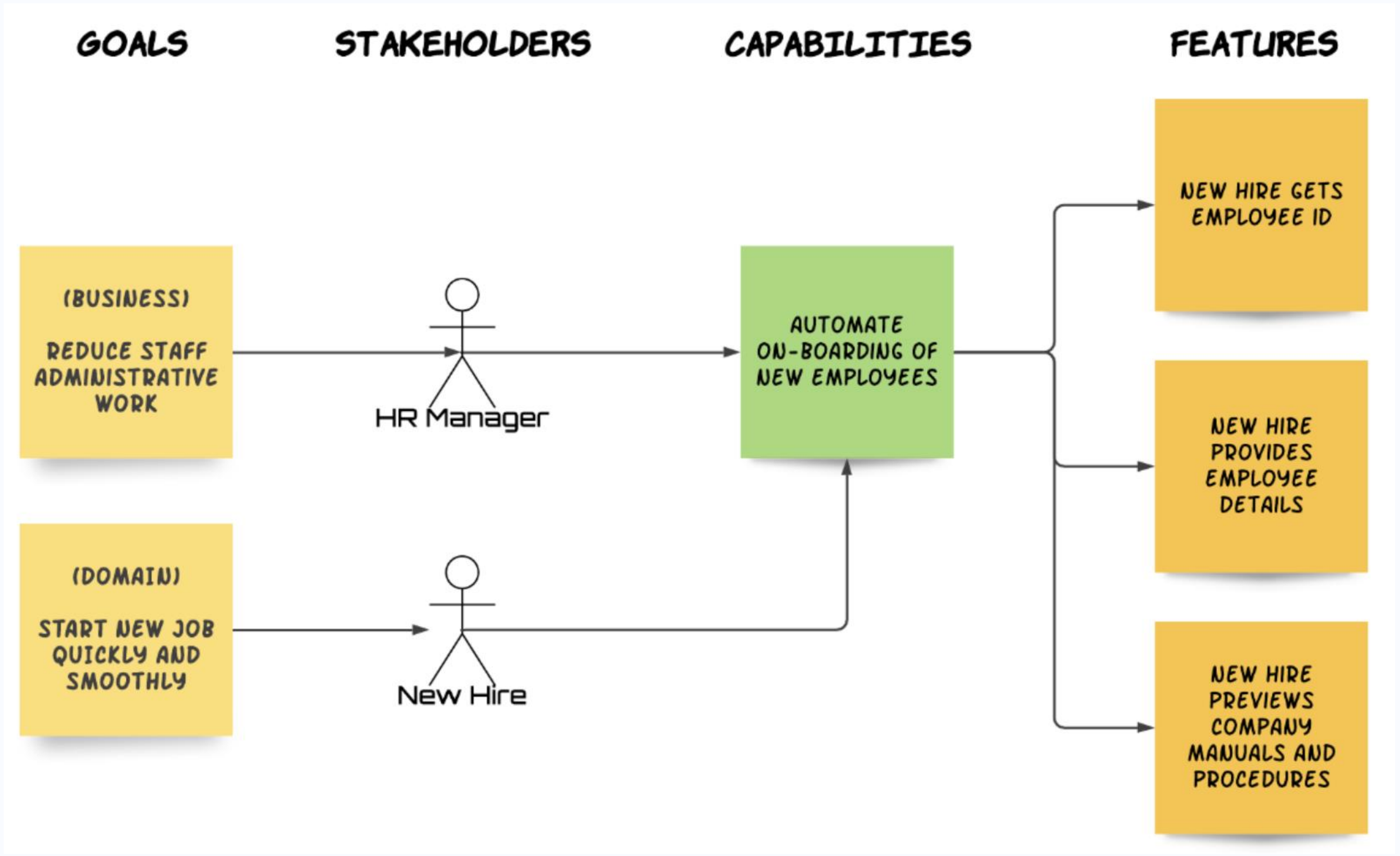
Use case 2 – onboarding new employees

- In this use case, we are working on a new Human Resources (HR) management system for our organisation.
- Our HR director tells us that they want our new HR system to facilitate the onboarding of new employees.
- The onboarding business process requires new hires to be allocated a company identification number, to provide some personal information, and be given the company manuals and procedure documents.
  - We can deduce (and confirm with the HR director) that the business goal they want to accomplish here is to automate the onboarding process so as to save staff time and company money.
  - Onboarding new employees is a capability because it is coarsely granular and describes something that the actor needs to be able to do in order to achieve the reduce staff work goal.

Use case 2 – onboarding new employees (Cont.)

- *Provide new employee with an employee ID*, on the other hand, captures a feature because it is very specific – it answers the question "h*ow can the system implement new employee onboarding?*" and doesn't by itself produce a tangible result.
- The same reasoning can be applied to conclude that *preview company manuals* and *procedures and capture employee details* are also features.
- We can model this analysis as an impact map in the following diagram:

Requirements modelling for the HR management system


GOALS
STAKEHOLDERS
CAPABILITIES
FEATURES
(BUSINESS) REDUCE STAFF ADMINISTRATIVE WORK
(DOMAIN) START NEW JOB QUICKLY AND SMOOTHLY
HR Manager
New Hire
AUTOMATE ON-BOARDING OF NEW EMPLOYEES
NEW HIRE GETS EMPLOYEE ID
NEW HIRE PROVIDES EMPLOYEE DETAILS
NEW HIRE PREVIEWS COMPANY MANUALS AND PROCEDURES

University of Southampton

Some notes

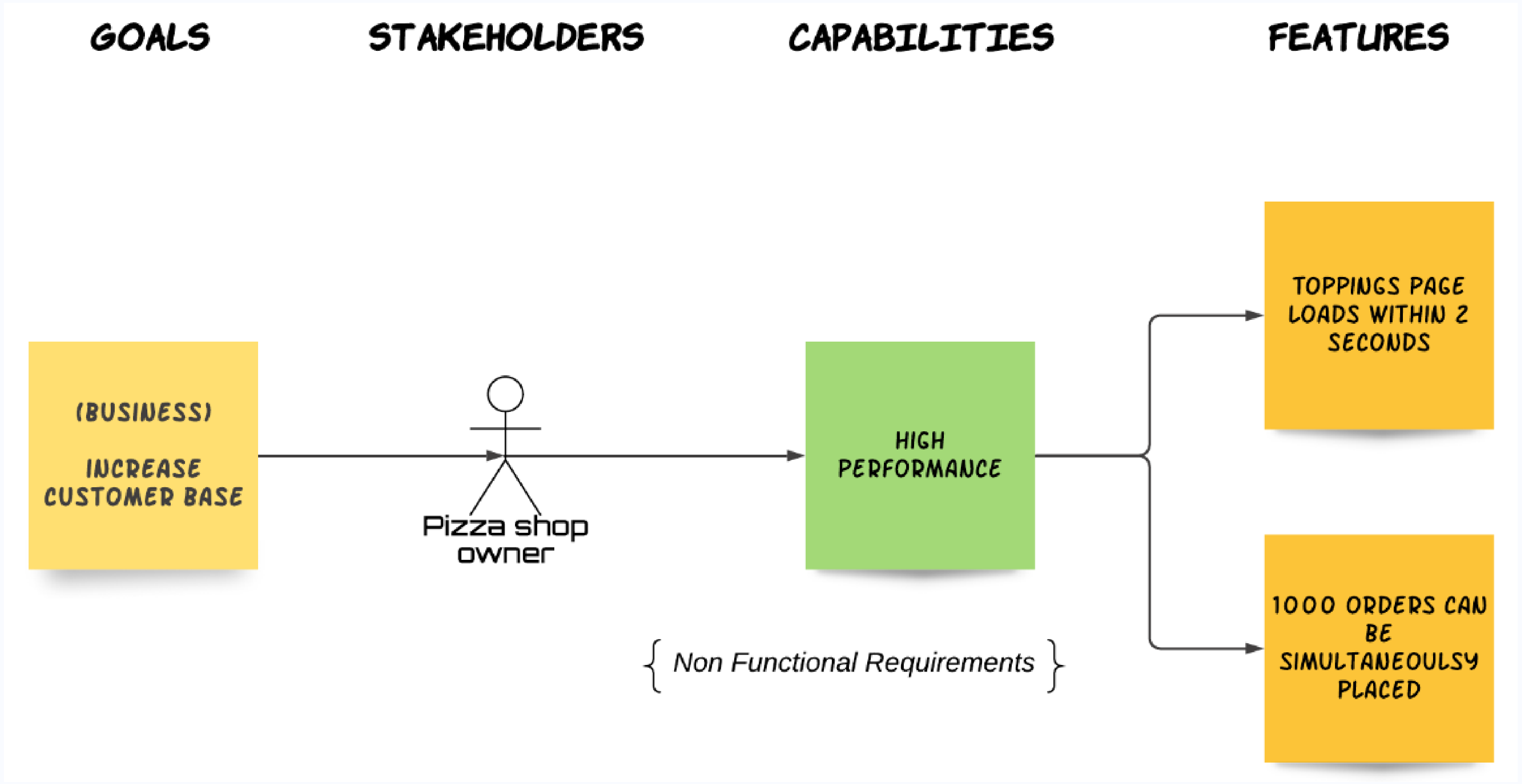
- You may notice that we have also defined a goal for our new hire actor.
  - The same capability often serves the purposes of more than one stakeholder. These can be both actors and non-acting stakeholders,
  - It can help achieve both business and domain goals (we talked about different goals in the Identifying goals section of Chapter 1, The Requirements Domain).
- The HR manager needs to have this capability available to system users in order to reduce HR staff administrative work.
- The new hire needs this capability for themselves in order to start their new working experience as quickly and smoothly as possible
  - It is important to identify other stakeholders and goals associated with a capability. This helps us discover more capabilities, features, and stakeholders that we might not have otherwise discovered.

Functional and non-functional requirements

- There are many way to classify our requirements. One popular approach is Functional and Non-Functional Requirements (NFRs)
- Functional requirements define what a product must do and what its features and functions are.
    - Examples: Creating user profiles, Logging workouts, Tracking progress, Displaying work-out details, Sending notifications about upcoming workouts
- Nonfunctional requirements describe the general properties of a system. They are also known as constraints or quality attributes.
    - Examples: How easy it is to use, How well it runs, How reliable it is
    - NFRs are very important as they often are what *Service Level Agreements (SLAs)* are based on. SLAs usually specify constraints for the system's *availability*, *performance*, and *responsiveness*.

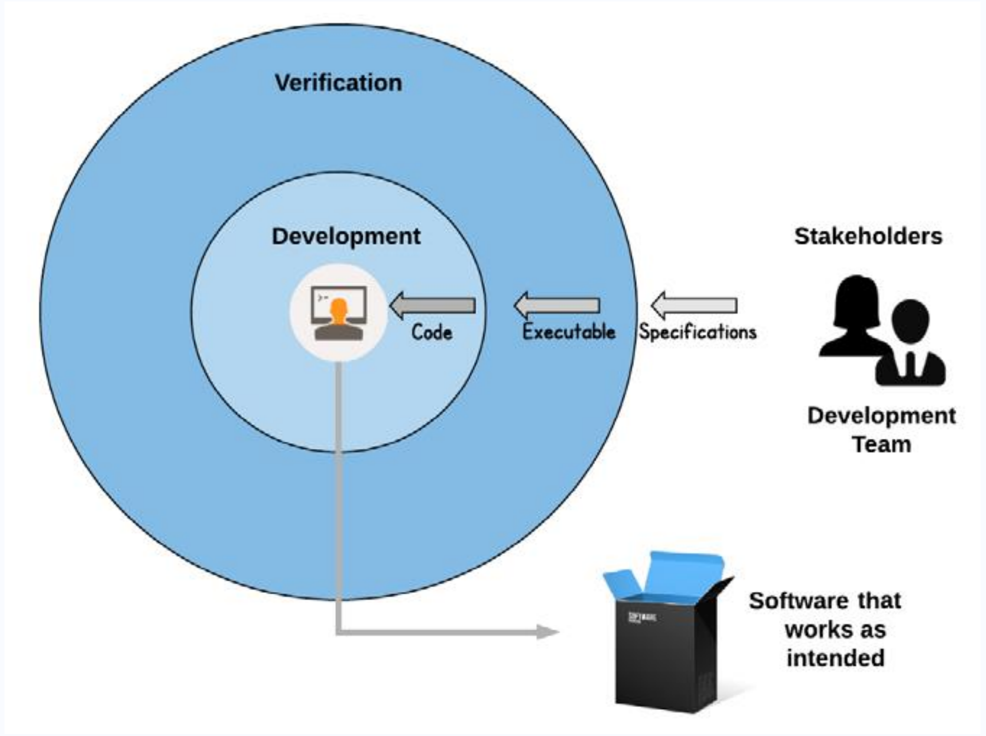
University of Southampton

Capturing Nonfunctional requirements with impact map


GOALS
STAKEHOLDERS
CAPABILITIES
FEATURES
(BUSINESS) INCREASE CUSTOMER BASE
Pizza shop owner
HIGH PERFORMANCE
TOPPINGS PAGE LOADS WITHIN 2 SECONDS
1000 ORDERS CAN BE SIMULTANEOULSY PLACED
{ Non Functional Requirements }

Requirement Specification – BDD

- What is BDD?
  - BDD (behaviour-driven development) is a software development process based on the Agile methodology.
  - The concept of behaviour-driven development originated from the test-driven development (TDD) approach.
  - It focuses on system behaviour required by the stakeholders as the driving mechanism for developing software. This well-defined system behaviour is referred to as a feature in BDD parlance.



The stakeholders and the development team come together to create the **specifications**. The specifications are, in effect, our features as described in the *What is a feature?*

BDD with impact mapping – a perfect partnership

- By looking at the BDD diagram, you may have noticed that the BDD life cycle begins with the establishment of some specifications, which is the discovery and creation of some features.
- However, BDD does not prescribe any specific way of discovering these features.
- Most BDD books and proponents advocate different ways of formulating user stories after communicating with the stakeholders.
- However, leveraging user stories as a means of capturing requirements and translating them into specifications can be vague, risky, and confusing.

Elements of feature specification

A fully formed feature will include the following:

- Feature Title: A brief description of the presented functionality – for example, ***Bank customer withdraws cash from cash machine***.
- User Story: Most people use the following template:
  - As an [`Actor`]
  - I want [`specific system behavior`]
  - So as to achieve [`a goal contributing to a Capability`]
- Impact: A link to the impact map this feature relates to.
- Notes: Any other text that helps the reader better understand the feature.
- Background (if applicable): A prerequisite or condition common across all scenarios.

Elements of feature specification – Cont.

- **Scenarios**: A descriptive feature will have *several scenarios*, which are written in a structured manner:
  - Given `[a Condition]`
  - When `[an Action is invoked]`
  - Then `[an expected Outcome occurs]`

The qualities of effective features specification

- Features serve two purposes:
  - They specify system behaviour in a clear and structured manner so that they can be read and understood by any and all stakeholders.
  - They allow system behaviour to be verified against system releases and deployments by using automation tools that match system behaviours to verification code.
- If a stakeholder cannot – or will not – read a *Feature* because it is too long or complicated, or contains technical jargon, then that feature is useless.
- The main purpose of a feature is to be read. If the structure or language of the feature hinders or prevents readability, then that feature is not fit for purpose!
- Features are not written as free or unstructured text. We write Features using a specific structure and following certain rules. In fact, there is a whole domain language just for writing Features.

University of Southampton

Writing Features with Gherkin

- We write Features in a structured manner, using a natural language subset called **Gherkin** (`https://cucumber.io/docs/gherkin/reference/`).
- Gherkin documents, such as a feature file, are written in a specific syntax.
- Most lines in a Gherkin document start with a ***keyword***, followed by our own text. These keywords are as follows:
  - `Feature`
  - `Rule` (as of Gherkin version 6)
  - `Scenario` (or example)
  - `Given`, `When`, `Then`, `And`, `*`
  - `Background`
  - `Scenario Outline` (or `Scenario Template`)
  - `Example`

Gherkin – Other features

- Comments are only permitted at the start of a new line, anywhere in the feature file.
- They begin with zero or more spaces, followed by a hash sign (#) and some text.
- Gherkin supports over **70 languages**, from Arabic to Uzbek, so we can write our Features in any language we choose.

A Feature's outline

```
Feature: My Beautiful Feature # The Feature title


## We can write anything we want from here until the next
keyword
## As suggested in the previous section we should put here the
following:
## -- User Story: a description of our feature
## -- Impact: a link to our Impact Map for this feature
## -- Notes to help readers understand the Feature


Background: # (optional) a common Condition which applies to
all scenarios
Scenario: # a specific behavior of our Feature
Scenario: # another behavior of our Feature
## …more Scenarios
```

University of Southampton

Variation in Feature's behaviour

- Feature should tell its readers all that they need to know in order to understand how the system will behave when that specific functionality is exercised.
- *Variation in Feature's behaviour* is described by using different **Scenarios**.
- **Scenarios** are written in the following manner:

```
Scenario: The Scenario Title
Given <a Condition>
And <another Condition>
And ….
When <an Event or Action takes place>
And <another Event or Action takes place >
And ….
Then <an expected Outcome occurs>
And <another expected Outcome occurs >
And ….
```

More on `Scenario`

- A `Scenario` has a title.
- As `Feature` may comprise many `Scenarios`, it is important that each `Scenario` has a descriptive title.
- A Scenario specifies system behaviour in terms of `Conditions`, `Events`, or `Actions` and `Outcomes`.
- Each `Condition`, `Event/Action` and `Outcome` consists of one or more **steps**, described in a single line of text.
- Each step is atomic; that is, it specifies a single `Condition`, `Event/Action`, or `Outcome` in its entirety.

More on `Scenario`

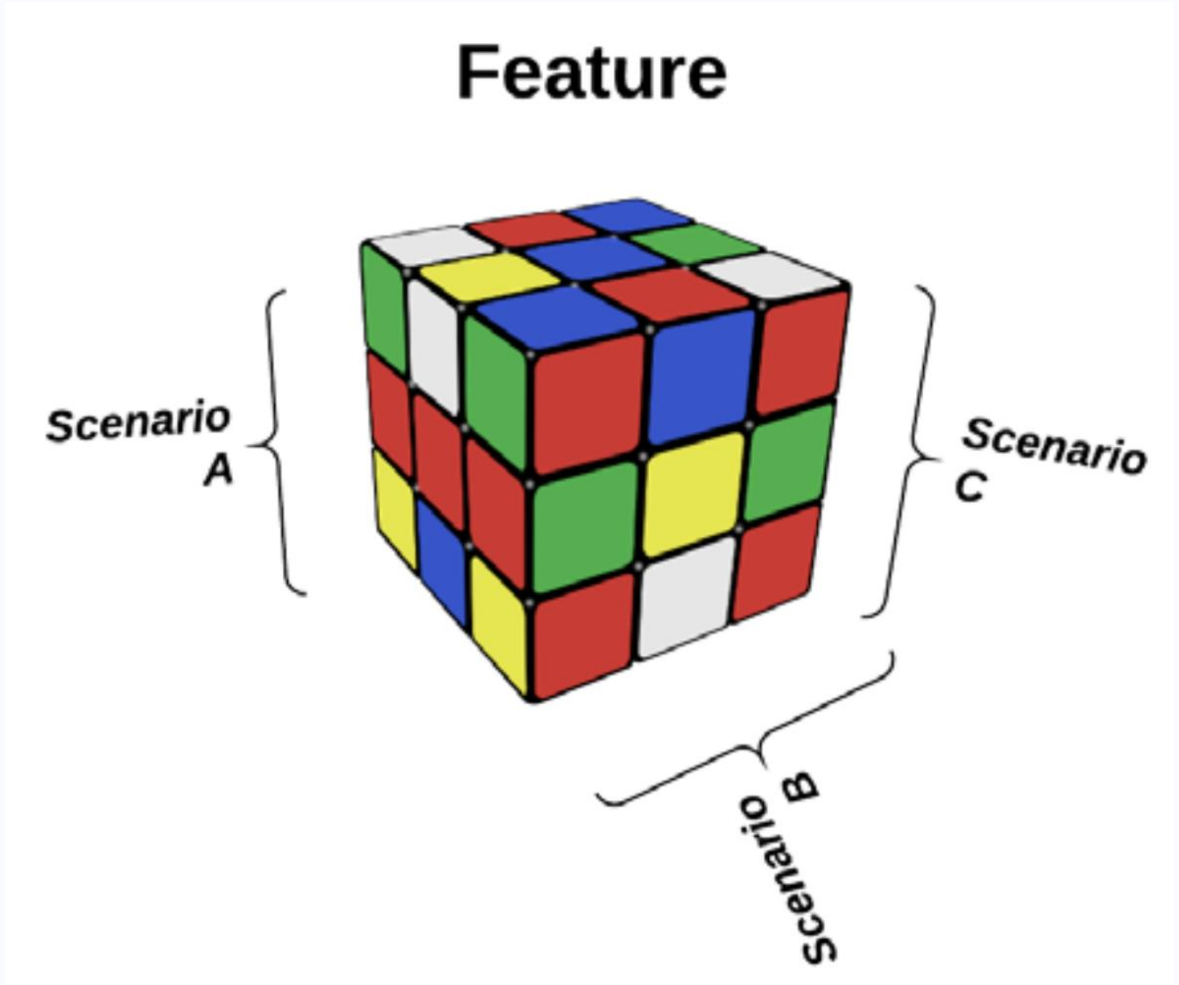
- To specify multiple `Conditions, Events/ Actions, or Outcomes,` we may use conjunctions such as *And*, *Or*, and *But* at the beginning of a new step.
- All `Scenarios` are structured in the same way: `Given` some conditions, `When` certain events or actions take place, `Then` specific outcomes should occur.
- This structure makes it easy to accurately specify our system's behaviour while also facilitating the creation of executable steps.

Scripting Scenarios

- Scenarios are a feature's essence.
- They reflect the change in the feature's behaviour under different circumstances.
- We call this section Scripting Scenarios as Scenarios are written similarly to a stage play or film script.
- They are written using prompts and specific actions for particular actors.
- Before we discuss how to script our Scenarios, let's see how we can discover them first.

Discovering Scenarios

- Imagine that your feature is a Rubik's cube.
- You have it in your hands, and you turn it around, looking at it from different angles.
- You notice how each side has different colours in different arrangements.
- It's the same cube, but each time you turn it, you discover some new image patterns and cell arrangements:



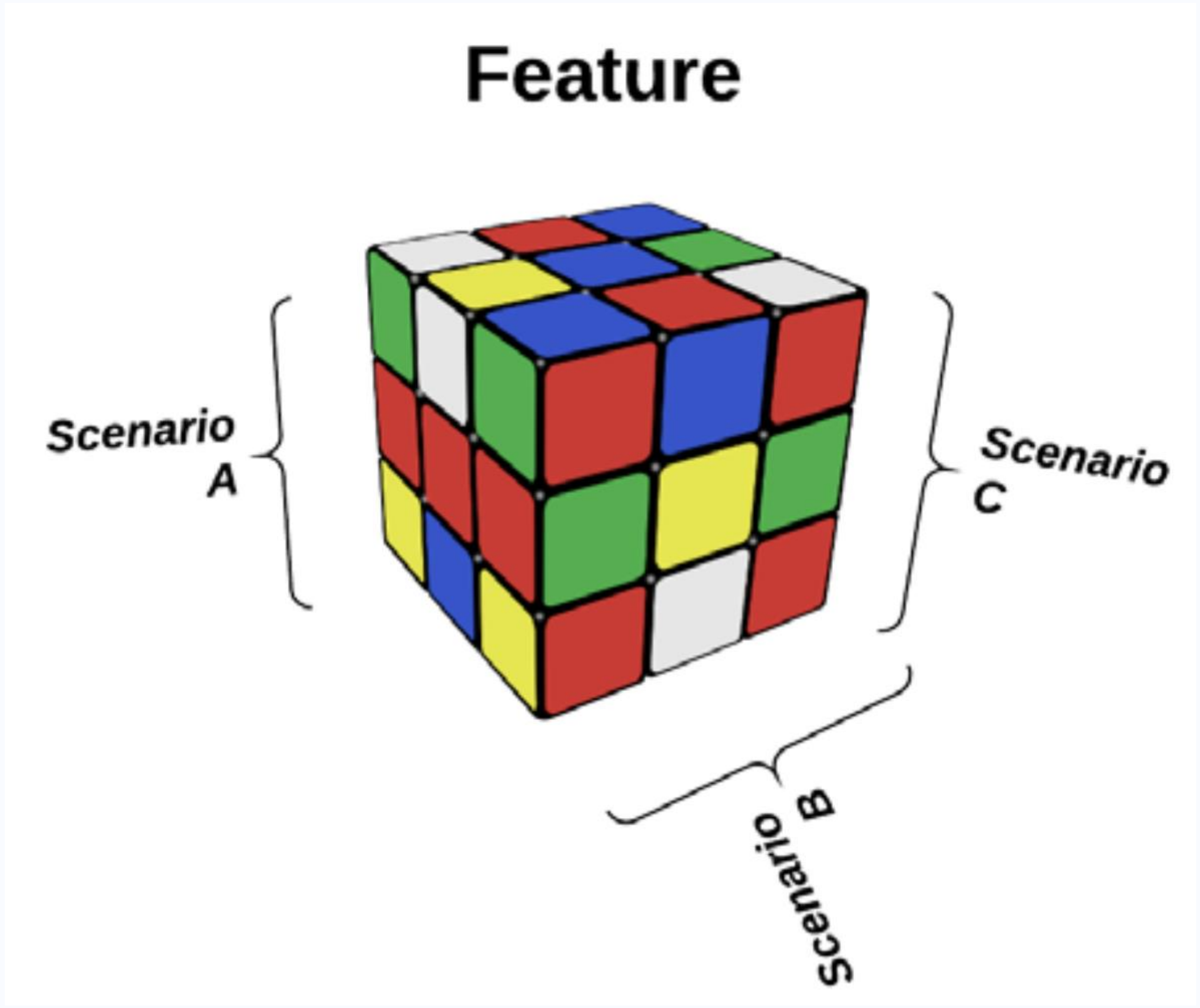
Feature and Scenarios

- When discovering *Scenarios*, we follow a similar mental process.
- We look at our *Feature* from *different angles* and *perspectives*.
- Feature itself *doesn't change*; it remains a *piece of system functionality* that *contributes* toward a *capability*.
- What changes is *how* that *functionality adapts* to different *circumstances*.

Writing Features with Gherkin. (Reminder)

- We write Features in a structured manner, using a natural language subset called **Gherkin** (`https://cucumber.io/docs/gherkin/reference/`).
- Gherkin documents, such as a feature file, are written in a specific syntax.
- Most lines in a Gherkin document start with a ***keyword***, followed by our own text. These keywords are as follows:
  - `Feature`
  - `Rule` (as of Gherkin version 6)
  - `Scenario` (or example)
  - `Given`, `When`, `Then`, `And`, `*`
  - `Background`
  - `Scenario Outline`(or `Scenario Template`)
  - `Example`

Feature and Scenarios – Reminder



University of Southampton

Feature and Scenarios

- When discovering *Scenarios*, we follow a similar mental process.
- We look at our *Feature* from *different angles* and *perspectives*.
- Feature itself *doesn't change*; it remains a *piece of system functionality* that *contributes* toward a *capability*.
- What changes is *how* that *functionality adapts* to different *circumstances*.

An Example

- Feature Title: The author uploads a picture to their profile.
- *Note1*: The author needs to have a profile picture so that readers can relate to them on a more personal level.
- *Note2*: Every Feature will have at least one ***Happy Path Scenario***.
  - This is the Scenario where the Feature's main actor accomplishes the feature's objective without any hindrance.
  - Some people call this the blue sky or best-case scenario.
- In our Feature example, the Happy Path Scenario may look like this:

Happy Path Scenario

```
Scenario: Happy Path
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
When the Author selects an image file from their computer
And the Author uploads the file
Then the Author sees their uploaded image as their Profile
picture
```

Happy Path Scenario

- This is a *Scenario* where all our assumptions were correct.
- "Which assumptions were these? "
  - Well, in this Scenario, we implicitly assumed the following:
    - The Author will be uploading a file stored on their computer.
    - The Author can choose any type of image file.
    - The system allows any size of image file to be uploaded.
- However, after consulting with our development team, we find out that not all image file types can be treated the same way (for various technical reasons).
- Furthermore, we are told that high-resolution images take up a lot of storage and should be avoided.

University of Southampton

Re-writing our Happy Path Scenario based on constraints

```
Scenario: Successful image upload
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
When the Author selects an image file from their computer
And the file is of type:
| file-type |
| jpg |
| gif |
| png |
And the file is of a size less than  "5 " MB
And the file is of a resolution higher than  "300 " by  "300 "
pixels
And the Author uploads the file
Then the Author sees their uploaded image as their Profile
picture
```

Some Notes

- When we re-write our *Happy Path Scenario* **Non-Functional Requirements (NFRs)** trickle into our features and start influencing our scenarios.
- When we look at a Happy Path Scenario and try to imagine what happens when our assumptions and constraints do not apply, we say that we invert the Scenario.
  - Scenario inversion is a great way to discover more Scenarios.
  - You may have noticed that we changed our Scenario's title from the generic ***Happy Path*** to the more descriptive ***Successful image upload***.
  - We sometimes get to have more than one Happy Path Scenario within our Feature, so it's good practice to give each scenario a meaningful name.

Avoiding repetition with Data Tables

- When we specified the file type step in our preceding scenario, we used a list with the acceptable file types.
- This list is actually a one-column table.
- The Gherkin language allows us to do that so that we can avoid the repetition of steps.
- We could have said this:

```
And the file type is jpg
Or the file type is gif
Or the file type is png
```

We used a **Data Table** instead:

```
And the file is of type:
| file-type |
| jpg |
| gif |
| png |
```

- Using Data Tables makes our scenario neater and easier to read.

Adding more Scenarios

- Now let's capture our system's behaviour when those constraints are broken, by adding some more Scenarios:

```
Scenario: Wrong type of image
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
When the Author selects an image file from their computer
And the file is of type:
| file-type |
| svg |
| tiff |
| bmp |
Then the Author sees a message informing them that the file
type is not supported
And the file is not uploaded
```

Adding more Scenarios

```
Scenario: Image too large
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
When the Author selects an image file from their computer
And the file is of type:
| file-type |
| jpg |
| gif |
| png |
And the file is of size greater than  "5 " MB
Then the Author sees a message informing them that the file is
too big
And the file is not uploaded
```

University of Southampton

Adding more Scenarios – Trying to upload large files

```
Scenario: Image too large
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
When the Author selects an image file from their computer
And the file is of size greater than  "5 " MB
Then the Author sees a message informing them that the file is
too big
And the file is not uploaded
```

In this specific Scenario, we don't care about file types. The only criterion for failure here is the file size.

Avoiding repetition with Scenario Outlines

```
Scenario Outline: Resolution too low
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
When the Author selects an image file from their computer
And the file is of type:
| file-type |
| jpg |
| gif |
| png |

And the file is of a resolution <width> pixels by <height>
pixels
# anything below 300 pixels won't be clear enough
Then the Author sees a message informing them that the file
resolution is too low
And the file is not uploaded
Examples:
| width | height |
| 200| 300 |
| 300| 200 |
| 299| 301 |
| 299| 299|
```

Some Notes

- We started our `Scenario` with the `Scenario Outline` keyword, instead of the `Scenario` keyword. This tells our readers (and also any BDD tools such as Cucumber) that this isn't a single scenario but a multi-value scenario.
- At the end of our `Scenario`, we defined an `Examples` table, with appropriate headers (width and height). We then defined a combination of different values of width and height that will incur a *failure* to upload outcome.
- In our steps, we used parameters delimited with `<>`, which reference the headers in the `Examples` table.
- In practical terms we compressed multiple `Scenarios` into one. Cucumber, JBehave, and so on will run this `Scenario` multiple times, once for each data row in the `Examples` table.
- `Scenario Outlines` are all about iteration and parameterization.

Avoiding step repetition with Backgrounds

- Another way to avoid repetition in our Scenarios is to use a **Background**.
- A `Background` is simply an abstraction of steps that are duplicated across multiple `Scenarios`.
- In our example Feature in the previous slides, every single `Scenario` begins with the same steps:

```
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
```

- We can abstract these steps in a `Background`, so as to avoid repeating them in all our `Scenarios`:

```
Background:
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
```

Some notes on `Background`

- Backgrounds are placed at the top of our Feature, before any `Scenarios`.
- Semantically, when we see a `Background`, we understand that the steps within it will be applied to every single `Scenario` of our Feature.
- So, now that we know how to write `Features`, let's see what a complete `Feature` looks like...

University of Southampton

Writing a fully formed Feature

```
Feature: Author uploads picture to profile

User Story: As an Author, I want to publish a nice picture in
my Profile, so that readers can relate to me on a human level
Impact: http://example.com/my-project/impacts-map
Screen mockups: http://example.com/my-project/mockups
Notes: Ms Smith, the CTO, really wants the Authors to upload
some nice photos of them to their profile as it helps sell more
books

Background:
Given the user is logged in as an Author
And the Author goes to their Profile page
And the Author chooses to upload a picture
And the Author selects an image file from their computer
```

University of Southampton

Successful Scenario

```
Scenario: Successful image upload
When the selected file is of type:
| file-type |
| jpg |
| gif |
| png |
And the file is of a of a size less than  "5 " MB
And the file is of a resolution higher than  "300 " by  "300 "
pixels
And the Author uploads the file
Then the Author sees their uploaded image as their Profile
picture
```

University of Southampton

Scenario for broken file type and size constraints

```
Scenario: Wrong type of image
When the selected file is of type:
| file-type |
| svg |
| tiff |
| bmp |
Then the Author sees a message informing them that the file
type is not supported
And the file is not uploaded
Scenario: Image too large
When the selected file is of type:
| file-type |
| jpg |
| gif |
| png |
And the file is of size greater than  "5 " MB
Then the Author sees a message informing them that the file is
too big
And the file is not uploaded
```

Scenario for broken file resolution constraint

```
Scenario: Resolution too low
When the selected file is of type:
| file-type |
| jpg |
| gif |
| png |
And the file is of size less than  "5 " MB
And the file is of a resolution:
# anything below 300 pixels won't be clear enough
| width | height |
| 200| 300 |
| 300| 200 |
| 200| 200 |
Then the Author sees a message informing them that the file
resolution is too low
And the file is not uploaded
```

Notes on a fully formed Feature

- Our `Feature` has a *title* and *contextual information* (`Notes`, `User Story`, `Impact Map`, link, and UI mock-ups).
- It details a *successful Scenario* and also its inversions (remember the Rubik's cube analogy?).
- We say that such a `Feature` is a **fully formed** `Feature`.
- A well-written `Feature` that covers all the main behaviours in four distinct `Scenarios`

Tips for writing good Features

- Features may seem easy to write but writing good `Features` requires focus and attention to detail. Here are a few observations:
- Title: In our Happy Path Scenario, we indicated what the behaviour in this Scenario is about (image upload) and its outcome (successful upload).
  - In the Scenarios that described unsuccessful behaviours, we described the causes of failure (Resolution too low, Image too big) in the titles.
- Actors: One of the first things to do when writing `Features` is to identify the role our actor has in this particular `Feature`.
  - In our Background, we clarified the actor's role (Given the user is logged in as an Author). This is very important as system behaviour is greatly affected by the actor's role and permissions.

Tips for writing good Features – Cont.

- It is crucial that throughout our `scenarios`, we know exactly under what guise the actor-system interactions take place.
- We should avoid referring to a generic user (for example, the user does something) but instead explicitly specify the Actor's role.
- We use generic and abstract language to describe our steps.
  - We avoid using technology or interface-specific language.
  - So, we never say that the actor clicks on the submit button to upload a picture.
  - We say instead that the actor chooses to upload a picture.
  - We don't state that the actor brings up the File Dialog to select an image file.
  - We instead say that they select an Image File from their computer.

Tips for writing good Features – Cont.

- We do that to separate ***system behaviour*** from ***system implementation***.
  - Today, the actor uploads a picture by clicking on a button.
  - Tomorrow, they will be uploading the picture by selecting it from a sidebar widget.
  - Next week, they may be uploading pictures by voice commands.
- The point is, the system's implementation of a behaviour will change much more frequently than the system behaviour itself.
- Always describe behaviours using generic *action verbs* rather than specific technical details.

Tips for writing good Features – Cont.

- Use imperative, present tense verbs to show behavior.
  - We say that the author selects an image, not that the Author can select an image or will select an image.
- When we write our `Scenarios`, we imagine the actions happening before us; there is no optionality or delay involved.
- Scenarios: Our `Scenarios` are atomic.
  - This means that each `Scenario` is executed completely independently from others.
  - `Conditions` needed for a `Scenario` must be specified in their entirety in the `scenario's condition (When)` steps or in the `Feature`'s `Background`.
  - No `scenario` should rely on `conditions` set by a different `scenario`.

Further reading

1. Gojko Adzic, *Impact Mapping: Making a Big Impact with Software Products and Projects*, ISBN-10: 0955683645
2. Dan North, *Introducing BDD*: `https://dannorth.net/introducing-bdd`
3. John Ferguson Smart, *BDD in Action: Behavior-driven development for the whole software lifecycle, Manning Publications, 1st edition*, ISBN-10: 161729165X
4. Gojko Adzic, *Specification by Example: How Successful Teams Deliver the Right Software, Manning Publications, 1st edition*, ISBN-10: 1617290084
5. Mike Cohn, *User Stories*: `https://www.mountaingoatsoftware.com/ agile/user-stories`

YOUR QUESTIONS