Requirements and User Stories(III)

Dr. Elham Mahmoudzadeh

Isfahan University of Technology

mahmoudzadeh@iut.ac.ir

2024

INVEST in Good Stories

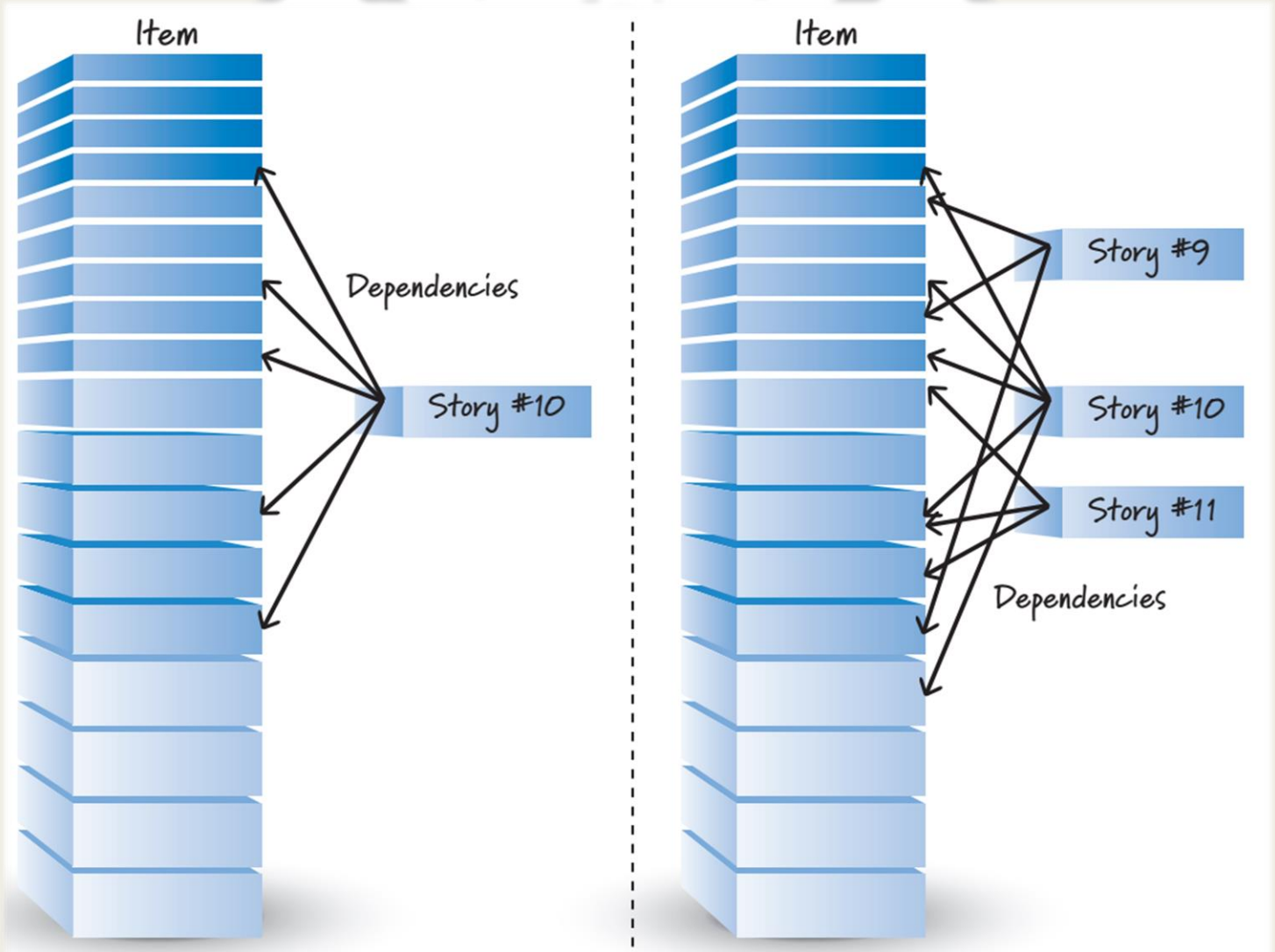
- How do we know if the stories that we have written are good stories?
- Six criteria (summarized by the acronym INVEST) that have been proved useful when evaluating whether our stories are fit for their intended use or require some additional work.

- The INVEST criteria are ***Independent***, ***Negotiable***, ***Valuable***, ***Estimatable***, ***Small*** (sized appropriately), and ***Testable***.

INVEST: Independent

- User stories should be *independent* or at least only loosely coupled with one another.
- Stories that exhibit a high degree of interdependence complicate estimating, prioritizing, and planning.
- When applying the *independent* criteria, the goal is not to eliminate all dependencies, but instead to write stories in a way that minimizes dependencies.

Highly dependent stories



Independent(Cnt'd)

- On the left side of Figure, story #10 depends on many other stories.
- Before we can work on story #10, we must first develop all of the dependent stories. In this single case that might not be so bad.
- However, imagine that you have many different stories with a high degree of interdependence, as illustrated by the right side of Figure.
- Trying to determine how to prioritize all of these stories and deciding which stories to work on in a sprint would be difficult to say.

INVEST: Negotiable

- The details of stories should also be *negotiable*.
- Stories are not a written contract in the form of an up-front requirements document.
- Instead, stories are placeholders for the conversations where the details will be negotiated.
- Good stories clearly capture the essence of what business functionality is desired and why it is desired.
- They leave room for the product owner, the stakeholders, and the team to negotiate the details.

Negotiable(Cnt'd)

- This negotiability helps everyone involved avoid the us-versus-them, finger pointing mentality that is commonplace with detailed up-front requirements documents.
- When stories are negotiable, developers can't really say, "Hey, if you wanted it, you should have put it in the document," because the details are going to be negotiated with the developers.
- And the business people can't really say, "Hey, you obviously didn't understand the requirements document because you built the wrong thing," because the business people will be in frequent dialogue with the developers to make sure there is shared clarity.

Negotiable(Cnt’d)

- Writing negotiable stories avoids the problems associated with up-front detailed requirements by making it clear that a dialogue is necessary.
- A common example of where negotiability is violated is when the product owner tells the team *how* to implement a story.
- Stories should be about what and why, not how.

Negotiable(Cnt’d)

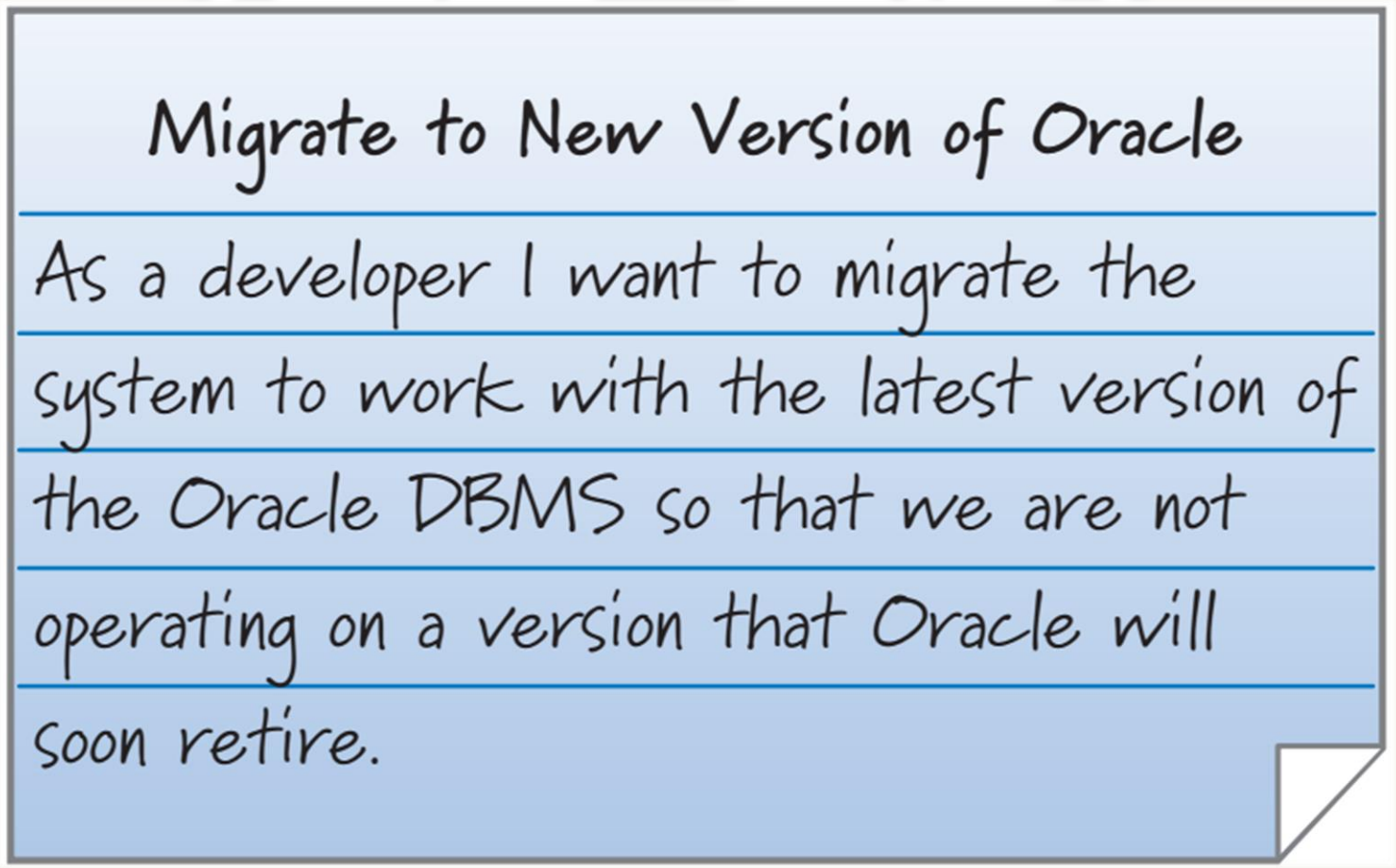
- There are times, however, when *how* something is built is actually important to the product owner. For example, there might be a regulatory obligation to develop a feature in a particular way, or there might be a business constraint directing the use of a specific technology. In such cases the stories will be a bit less negotiable because some aspect of the “how” is required.

Not all stories are fully negotiable, but most stories should be.

INVEST: Valuable

- Stories need to be *valuable* to a customer, user, or both.
- Customers (or choosers) select and pay for the product. Users actually use the product.
- If a story isn't valuable to either, it doesn't belong in the product backlog.
- We would either rewrite the story to make it valuable to a customer or user, or we would just discard it.
- How about stories that are valuable to the developers but aren't of obvious value to the customers or users? Is it OK to have technical stories.

Example technical story



Technical stories

- The fundamental problem with technical stories is that the product owner might not perceive any value in them, making it difficult if not impossible to prioritize them against business-valuable stories.
- For a technical story to exist, the product owner should understand why he is paying for it and therefore what value it will ultimately deliver.

Technical stories(Cnt'd)

- In the case of the "Migrate to New Version of Oracle" story, the product owner might not initially understand why it is valuable to change databases.
- However, once the team explains the risks of continuing to develop on an unsupported version of a database, the product owner might decide that migrating databases is valuable enough to defer building some new features until the migration is done.
- By understanding the value, the product owner can treat the technical story like any other business-valuable story and make informed trade-offs.
- As a result, this technical story might be included in the product backlog.

Valuable(Cnt'd)

- In practice, most technical stories should not be included in the product backlog.
- Instead, these types of stories should be tasks associated with getting business valuable stories done.
- If the development team has a strong definition of done, there should be no need to write stories like these.

All stories in the backlog must be valuable (worth investing in) from the product owner's perspective, which represents the customer and user perspectives.

Not all stories are independent, and not all stories are fully negotiable, but they all must be valuable.

INVEST: Estimatable

- Stories should be *estimatable* by the team that will design, build, and test them.
- Estimates provide an indication of the size and therefore the effort and cost of the stories (bigger stories require more effort and therefore cost more money to develop than smaller stories).

Estimatable(Cnt'd)

- Knowing a story's size provides actionable information to the Scrum team.
- The product owner, needs to know the cost of a story to determine its final priority in the product backlog.
- The Scrum team, on the other hand, can determine from the size of the story whether additional refinement or disaggregation is required. A large story that we plan to work on soon will need to be broken into a set of smaller stories.

Estimatable(Cnt'd)

- If the team isn't able to size a story, the story is either just too big or ambiguous to be sized, or the team doesn't have enough knowledge to estimate a size.
- If it's too big, the team will need to work with the product owner to break it into more manageable stories.
- If the team lacks knowledge, some form of exploratory activity will be needed to acquire the information.

INVEST: Sized Appropriately (Small)

- Stories should be *sized appropriately* for when we plan to work on them.
- Stories worked on in sprints should be *small*.
- If we're doing a several-week sprint, we want to work on several stories that are each a few days in size.
- If we have a two-week sprint, we don't want a two-week-size story, because the risk of not finishing the story is just too great.

Sized Appropriately (Cnt'd)

- So ultimately we need small stories, but just because a story is large, that doesn't mean it's bad.
- Let's say we have an epic-size story that we aren't planning to work on for another year. Arguably that story is sized appropriately for when we plan to work on it.
- In fact, if we spent time today breaking that epic down into a collection of smaller stories, it could easily be a complete waste of our time.
- Of course, if we have an epic that we want to work on in the next sprint, it's not sized appropriately and we have more work to do to bring it down to size.
- You must consider *when* the story will be worked on when applying this criterion.

INVEST: Testable

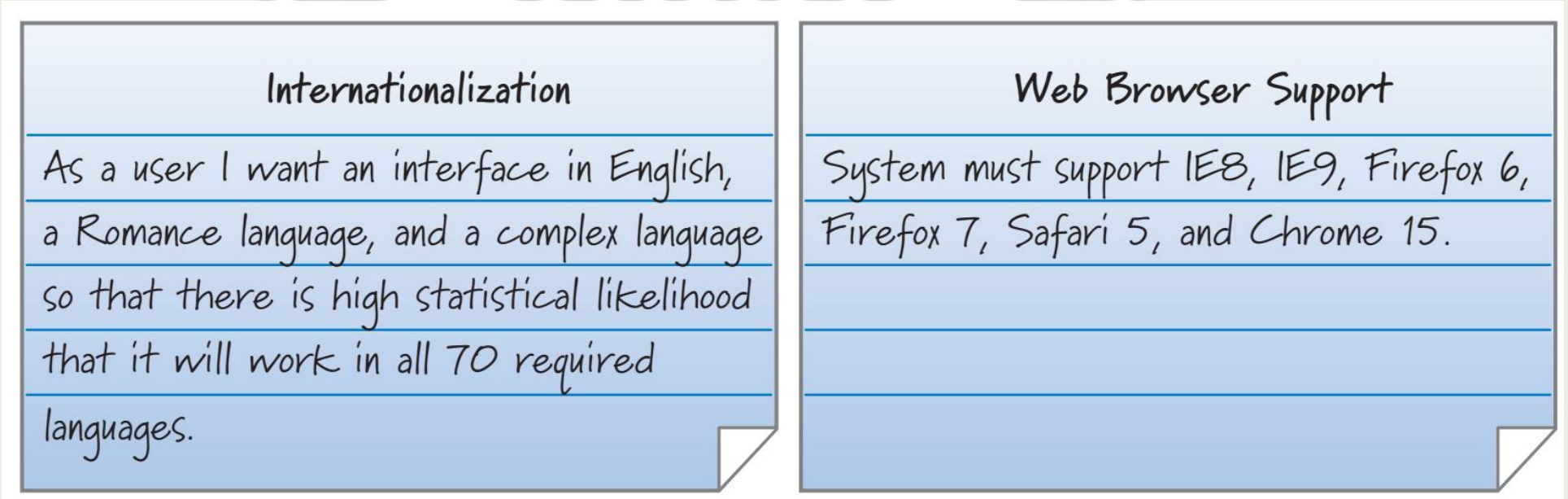
- Stories should be *testable* in a binary way—they either pass or fail their associated tests.
- Being testable means having good acceptance criteria (related to the conditions of satisfaction) associated with the story, which is the "confirmation" aspect of a user story.
- Without testable criteria, how would we know if the story is done at the end of the sprint?
- Also, because these tests frequently provide important story details, they may be needed before the team can even estimate the story.

Testable(Cnt'd)

- It may not always be necessary or possible to test a story.
- For example, epic-size stories probably don't have tests associated with them, nor do they need them (we don't directly build the epics).
- Also, on occasion there might be a story that the product owner deems valuable, yet there might not be a practical way to test it.
- These are more likely to be nonfunctional requirements, such as "As a user I want the system to have 99.999% uptime." Although the acceptance criteria might be clear, there may be no set of tests that can be run when the system is put into production that can prove that this level of uptime has been met.

Nonfunctional Requirements

- Represent system-level constraints.



Nonfunctional Requirements(Cnt'd)

- As system-level constraints, nonfunctional requirements are important because they affect the design and testing of most or all stories in the product backlog.
- For example, having a "Web Browser Support" nonfunctional requirement would be common on any website project. When the team develops the website features, it must ensure that the site features work with all of the specified browsers.

Nonfunctional Requirements(Cnt'd)

- The team must also decide when to test all of the browsers.
- If the team includes the "Web Browser Support" nonfunctional requirement in the definition of done, the team will have to test any new features added in the sprint with all of the listed browsers. If it doesn't work with all of them, the story isn't done.
- Try to include as many of the nonfunctional requirements in their definitions of done as they possibly can.
- Waiting to test nonfunctional requirements until late in the development effort defers getting fast feedback on critical system performance characteristics.

Knowledge-Acquisition Stories

- Sometimes we need to create a product backlog item that focuses on knowledge acquisition.
- Perhaps we don't have enough exploitable knowledge about the product or the process of building the product to move forward.
- Such exploration is known by many names: *prototype*, *proof of concept*, *experiment*, *study*, and so on.
- They are all basically exploration activities that involve buying information.
- Try to employ a user story as the placeholder for the exploration work.

Knowledge-acquisition story

Filtering Engine Architecture Eval

As a developer I want to prototype two alternatives for the new filtering engine so that I know which is a better long-term choice.

Conditions of Satisfaction

Run speed test on both prototypes.
Run scale test on both prototypes.
Run type test on both prototypes.
Write short memo describing experiments, results, and recommendations.

Example

- Team wants to evaluate two possible architectures for the new filtering engine.
- It is proposing to prototype both architectures and then run speed, scale, and type tests against both prototypes.
- The deliverable from the prototyping activity will be a short memo that describes the experiments that were performed, the results that were obtained, and the team's recommendation for how to proceed.

Knowledge-acquisition story (Cnt'd)

- This specific knowledge-acquisition story looks like a technical story.
- The business value of any technical story has to be justifiable to the product owner.
- Because product owners think in economic terms, there needs to be an economic justification for doing this prototyping work.
- There is likely a compelling technical argument for doing a knowledge-acquisition story because the team is typically blocked from making forward progress until it has the knowledge produced by the story.

Knowledge-acquisition story (Cnt'd)

- The question for the Scrum team is whether the value of the acquired information exceeds the cost of getting it.
- Here is how a Scrum team could approach answering that question.
- First, we need to know the cost of the prototyping.
- No good product owner will authorize unbounded exploration.

Knowledge-acquisition story (Cnt'd)

- The team might not be able to answer particular questions until an architectural decision has been made, but it must be able to answer the question of how much effort it wants to spend to buy the information necessary to make the architectural decision.
- So, we ask the team to size the prototyping story.
  - *Let's say that the size estimate indicates that the full team would need to work on the story for one sprint. We know who is on the team and the length of the sprint, so we also know the cost of acquiring the information. Let's say it is $10K.*

Knowledge-acquisition story (Cnt'd)

- Here is one way we might estimate the value. Imagine that I flip a coin. If it comes up heads, we'll do architecture A; if it comes up tails, we'll do architecture B.
- Now, I ask the team to estimate the cost of being wrong. For example, if I flip the coin and it comes up heads and we start building business features on top of architecture A, and architecture A turns out to be the wrong approach, what would be the cost to unwind the bad decision and rebuild everything on top of architecture B? Let's say the team estimates the cost to be $500K.

Knowledge-acquisition story (Cnt'd)

- Now we have enough information to make a sensible economic decision.
- Are we willing to spend $10K to purchase information that has an expected value of $250K (half the time we flip the coin we would be correct)?
- Sure, that seems like a sensible business decision.
- Now the product owner can justify why this story is in the backlog.

Knowledge-acquisition story (Cnt'd)

- What if the team's response to "What would it cost if we were wrong?" is $15K? In this case it would be a bad decision to do the prototyping story.
- Why spend $10K to buy information that has an expected value of $7.5K? We would be better off just flipping the coin (or making an educated guess) and, if we're wrong, simply redoing the work using the other architecture.
- It's an example of what some people call a **fail-fast strategy** (try something, get fast feedback, and rapidly inspect and adapt).

Reference

1- K. S. Rubin, “Essential Scrum, A Practical guide to the most popular agile process,” 2013.