



Hugbúnaðarverkefni 2 / Software Project 2

2. Agile Requirements

HBV601G – Spring 2019

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Update: Course Structure

■ Lectures

- Thursdays 08:30-09:50, Askja 132

■ Team consultations

- Weekly meetings with tutors
 - to discuss scope, design, progress
 - to present assignment deliverables
- Thursdays 13:20-16:30 (from 17 Jan), VR-II 152
 - Teams 1-10: 13:20-14:20
 - Teams 11-20: 14:25-15:25
 - Teams 21-30: 15:30-16:30

■ Team meetings

- Scheduled freely among team members

■ Project teams

- 3-4 students per team
- Recommended (but not required) to work in last semester's teams

■ Project topics

- e.g. a service, community, game, etc.
- Recommended (but not required) to continue working on same project

■ Basic architectural requirements

- Build a mobile app
- Native Android (not browser-based)
- Highly recommended to build on last semester's server-side code

Team Consultations (tentative)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
D	X			X			X			X			X							X			X			X				
A		X			X			X			X			X		X							X				X		X	
S			X			X			X			X			X										X			X		X
#								2									0	1	0		0	0					2			

■ Consultation time slots

- Thursdays 13:20-16:30, V-152
 - Teams 1-10: 13:20-14:20
 - Teams 11-20: 14:25-15:25
 - Teams 21-30: 15:30-16:30

■ Seating arrangement

- Daníel's 2019 teams: last rows
- Andri's 2019 teams: middle rows
- Sigurður's teams: first rows

■ Team/Tutor affiliations rotated

- To enable fresh feedback and avoid grading biases
- Table above reflects new affiliations (tentative)

■ Please let me know if you changed timeslots or formed a new team

- i.e. if more than half of your team members have a new team number

In-Class Quiz Prep

- Please prepare a scrap of paper with the following information:

ID: _____@hi.is	Date: _____
a) _____	d) _____
b) _____	e) _____
c) _____	f) _____

- During class, I'll show you questions that you can answer very briefly
 - Just numbers or letters, no elaboration
- Hand in your scrap at the end of class
- All questions in a quiz weigh same
- All quizzes (ca. 10-12 throughout semester) have the same weight
 - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam



Requirements in Agile Projects

see also:

Leffingwell: Agile Software Requirements, Ch. 6, 13, 19



Recap:

Valid Requirements Are the Key to Project Success

- The majority of project failures is due to incorrectly addressed requirements.
- Managing requirements is more challenging than technical execution of project.
- We can't just rely on the customers to tell us their requirements:
 - Customers often don't know exactly what they want
 - If they know what they want, they can't describe it precisely
 - If they can describe it, they describe a proposed solution rather than an actual need
 - If they describe an actual need, we still need to
 - understand it (what? how? why?)
 - explore it (variants? constraints? boundaries? exceptions? extensions? evolution?)
 - solve it (architecture? design? technology? dependencies? implementation? operation?)
- It is virtually impossible for customers or developers to have a correct and complete picture of all this right from the start of any project.

Recap:

Valid Requirements Are the Key to Project Success

- It is virtually impossible to develop valid requirements up front.
- Agile purist approach: “Let’s not attempt to define precise requirements then, but figure things out along the way.”
 - Taken literally, this just surrenders to the aforementioned problems, it doesn’t solve them.
- **Caution: Agile approaches do not make requirements work easier!**
 - In particular, requirements work does not disappear just because there is no document.
 - Agile approaches just interleave the requirements work more tightly with other activities and allow the team to make course corrections along the way.
 - i.e. they are in essence a risk management technique (since corrections would not be cheap)
- **Agile approaches require investing just as much thought and effort (and discipline) into the aforementioned issues as plan-driven approaches do.**
 - “Let’s just start coding and see what the customer says” can end up in an expensive disaster just as easily as “Let’s get the complete and precise requirements from the customer first.”

Defining and Communicating the Project Vision

- In agile approaches, we strive to prevent
 - overinvesting effort in things we are unlikely to understand very well initially anyway
 - binding resources too early on a fixed set of commitments that we may wish to change later
- No explicit project definition or Software Requirements Specification is prepared
- Need another way to communicate project vision to agile development team, i.e. to make developers aware of strategic intent as overall frame of reference:
 - Why are we building this product / system / application?
 - What problem will it solve?
 - What features and benefits will it provide? For whom?
 - What platforms, standards, applications etc. will it support?
 - What system landscape will it be integrated with?
- An explicit document (can be quite concise) is still the best way to express this

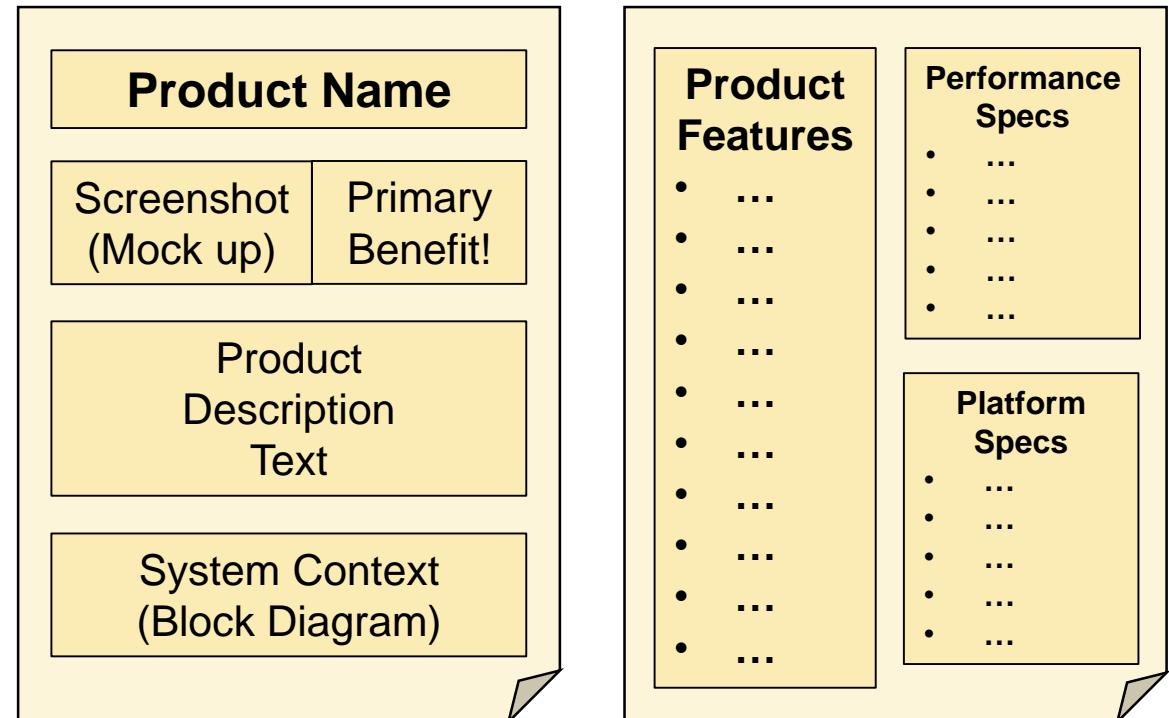
Defining and Communicating the Project Vision: Suitable Documents

a) Vision Document (→ HBV501G, Lecture 3)

- RUP Vision Document works well at the start of agile projects too (customize as needed)
- Encourages consideration of most relevant business and scope considerations

b) Product Data Sheet

- A two-page “flyer” highlighting product’s key characteristics
- Not just marketing-speak – describe product concisely and precisely
- Not intended for actual publication, but to encourage team to frame vision of final product



Defining and Communicating the Project Vision: Suitable Documents

c) Hypothetical Press Release

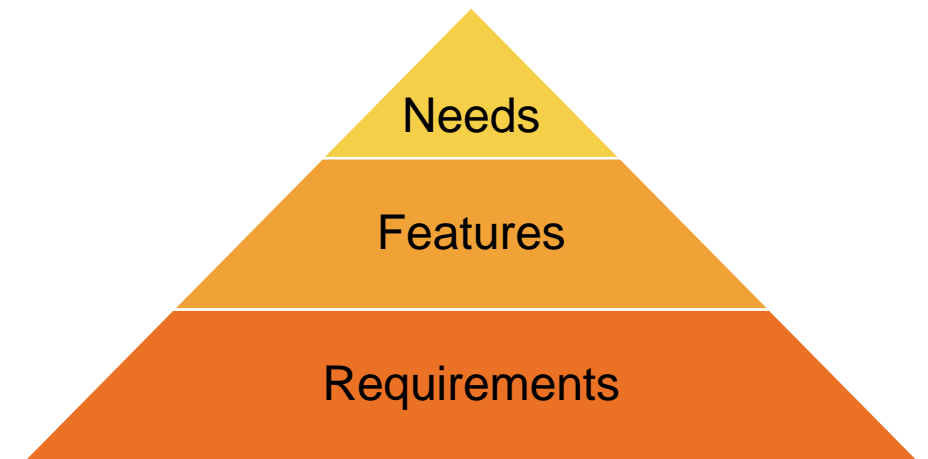
- Drafting hypothetical press release for the future release day can help teams to think the project through in user's and management's terms, and clearly articulate its benefits.
- Need to tell complex story in simple, concise way – opportunity to involve marketing dept. and entice internal stakeholders

d) Feature Backlog with Briefing

- If product managers or other responsible stakeholders have a clear enough idea of the project to begin elaborating a preliminary backlog, this can serve as communication basis
- Have dedicated vision workshop where preliminary backlog is introduced to team, together with discussion of the project motivation
- Preliminary backlog can serve as starting point for effort estimation and prioritization

Needs, Features and Requirements

- The project vision (in whichever form) focuses on **features**
 - i.e. capabilities provided by the system that fulfill one or more stakeholder needs
 - Described in concise key phrases, e.g. “spell checking as you type”
- Features bridge gap between the values that the system shall support (**needs**) and the exact form that this support shall take (**software requirements**)
 - Guideline: Any system (no matter how complex) can be described by < 15-25 key features
 - If you think you need more features, you’re thinking about them on a too detailed level
- Guideline for moving between abstraction levels:
 - First think and talk about **features** as an intuitive starting point for a product scope discussion
 - Then think and talk about **needs** to broaden scope and make sure you didn’t miss anything
 - Then think and talk about **requirements** to go into more detail about how features will be supported



Epics

- Another (optional!) abstraction level on top of features
 - In large projects, a common theme for several features (e.g. “Personalization”)
 - In large organizations, a driver for a particular IT project (e.g. “Move CRM to cloud”)
 - Use only if you feel you need to group your features into larger categories

Differences between abstraction levels:

	Epic	Feature	Requirement
Scope	Strategic direction, overarching business need	Value-delivering product capability supporting an epic	Concrete, functional materialization of part of a feature
Origin	Management, Enterprise Architects	Product Owner	Product Owner & Team
Timeframe	One or more releases	One or more iterations	One iteration
Format	Key phrase	Key phrase or user story	User story or use case
Location	Project vision	Project vision	Backlog

Example Epic, Needs and Features

- **Epic:**

- In addition to physical store, sell goods also in an online shop.

- **Needs:**

- Payment in the online shop needs to be effortless, so customers have no reason to change their mind and bail out of a purchase at the last second
- Customers need to trust the shop so they will return to it to buy more.
- The shopkeeper needs to minimize their risk of being defrauded by customers in order to remain profitable.
- [...]

- **Features regarding the need for “effortless payment”:**

- Payment by credit card
- Payment by PayPal
- Payment by invoice
- Payment by voucher

- **Features regarding the need for “trust in the shop”:**

- Notification in case of shipping delays
- Refund for returned goods
- Secure transmission of payment data
- Clear display of prices, terms and conditions

Example Requirements

■ Requirements for feature “Payment by credit card”:

- As a customer,
 - I can pay by credit card for my purchase so I don't need to remember making a payment before/after receiving the goods.
 - I can let the shop store my credit card information so I can pay more conveniently for subsequent purchases.
 - I can delete stored credit card information from the shop's records and thus remain in control of my personal data.
 - I am informed about the success or failure of each credit card transaction so I know if my payment went through.
- As a shopkeeper,
 - I want to accept payments by credit card so I can serve domestic and international customers using the same processes.
 - I can recognize invalid or blocked credit card numbers immediately so I don't need to spend fees on transactions that are bound to fail.
 - I want to receive confirmation of payment before shipping any goods, in order to reduce the risk of fraud.
 - I can reverse credit card payments in case of returns in order to keep my accounting simple.

In-Class Quiz #1: Types of Requirements



- Indicate if the following are (E)pics, (F)eatures, (N)eed, (R)equirements or (T)asks:
 - a) As a customer, I can let the shop store my credit card information so I can pay more conveniently for subsequent purchases.
 - b) Define acceptance test
 - c) In addition to physical store, sell goods also in an online shop.
 - d) Payment by credit card
 - e) Payment in the online shop needs to be effortless, so customers have no reason to change their mind and bail out of a purchase at the last second
 - f) Cancel an order

User Stories

- In agile projects, requirements are typically expressed as user stories:
 - “As a [role], I can [activity] so that [business value].”
- **User stories serve as a common communication format**
 - short, easy to read (→ *HBV401G, Lecture 3*)
 - not precise requirements specifications, but expressions of interest
 - understandable by and valuable to customers
 - understandable by and testable by developers
 - organized in lists that can be flexibly rearranged, rather than large documents
- Don't get too fixated on the above format though
 - It should encourage you to think about target groups, requirements and rationales
 - But if it gets cumbersome to express sth. this way, write it whichever way yields most clarity
 - e.g. “log in to my customer portal”, “see my daily data usage”, “check my phone bill”

User Stories

- In agile projects, requirements are typically expressed as user stories:
 - “As a [role], I can [activity] so that [business value].”
- **User stories represent “one unit of functionality”**
 - for all purposes of planning and measuring progress (→ *HBV401G, Lecture 5*)
 - small enough that the team can build about half a dozen in an iteration
 - small enough that one can be implemented in a period of days to weeks
 - Not detailed at the outset of the project but elaborated just-in-time
 - This means: You clarify them with the client *before* you start estimating and coding them.
 - This does not mean: You interpret them your way and then ask the customer if you guessed right.
- Besides “requirements” stories, a backlog can also contain other work items
 - e.g. refactoring, bug fixes, support, maintenance, tooling, infrastructure work
 - not written like a user story, but sized and treated as such for planning purposes

Refining User Stories

- Formats for expressing epics/needs, features, requirements are somewhat fluid
 - Prefer key phrases for the more high-level abstraction layers
 - Prefer user stories (or even use cases) for the lower abstraction layers
 - Most importantly: Write precisely and appropriately for the given abstraction layer
- Initial versions of user stories may be too broad
 - Some may actually not describe requirements, but features or even epics
 - Rephrase them as such, and consider what requirements would address that feature/epic.
 - Helps to structure requirements, and identify groups of requirements that are not sufficiently explored yet
 - Some may describe requirements too broadly
 - Split them up into several, more precise user stories
 - Helps to refine requirements and obtain more clearly defined, independently implementable functionalities
 - Stepwise refinement of user stories is natural, and a good thing
 - Great opportunity for focused discussion with customer and learning about the business domain

Patterns for Splitting User Stories

Initial user story	Split user stories
Split workflow steps	
As a phone company, I want to update and publish pricing programs to my customer.	...I can send a message to the customer's phone. ...I can show a message on the customer web portal. ...I can send an e-mail to the customer's account.
Make business rule variations explicit	
As a phone company, I can sort customers by different demographics.	...sort by ZIP code. ...sort by home demographics. ...sort by data usage.
Split according to major effort	
As a user, I want to be able to select/change my pricing program through my web portal.	...I want to use time-of-use pricing. ...I want to prepay for my data and voice usage. ...I want to enroll in critical-peak pricing.

First story includes infrastructural groundwork, others are simpler additions.

Patterns for Splitting User Stories

Initial user story	Split user stories
Focus on the simplest version that can work on its own	
As I user, I basically want a fixed price, but I also want to be notified of critical-peak pricing events.	...I want to be notified of critical peak pricing events. ...I want to be able to switch my plan temporarily.
Make procedural variations explicit	
As a phone company, I can send messages to customers.	...regarding their contract. ...regarding promotional offers. ...regarding general news.
Defer system qualities	
As a user, I can view my data usage in various graphs.	...using bar charts that compare weekly usage. ...in a comparison chart, so I can compare my usage to people from similar demographics.

Build a simple version first, and leave more complex story for later implementation.

Patterns for Splitting User Stories

Initial user story	Split user stories
Split operations	
As a user, I can manage my account.	...I can sign up for an account. ...I can edit my account settings. ...I can cancel my account. ...I can add more devices to my account.
Use case scenarios	
I want to transfer credits from the loyalty program of partner company X.	<i>Main scenario:</i> Notify companies of request and handle transfer: [...] <i>Alternative scenario:</i> Handle exceptions in the transfer process: [...]

Complex interactions may be more precisely and efficiently described as use cases.

Splitting User Stories

Initial user story	Split user stories
Research more to eliminate uncertainty (“spike”)	
As a consumer, I want to see my real-time and historic data usage visualized so that I can intuitively adjust quality-of-service parameters as needed.	<p>Technical spike: Research what level of real-time usage measuring and display is feasible, what degrees of QoS the company is willing to support, and how soon any user adjustments can take effect.</p> <p>Functional spike: Prototype a histogram in the web portal and get some user feedback on chart style and adjustment parameters.</p>

A spike is a special backlog item intended to eliminate uncertainty in a requirement.

Dealing With Uncertainty in Spikes

- **“Spikes”** are special backlog items aimed at reducing risk and uncertainty, e.g.
 - Performing basic research, familiarizing team with new technology or business domain
 - Analyzing a complex user story to see how it can be broken down into estimable pieces
 - Prototyping to gain confidence in a technology and make more informed estimate
- **Technical spikes** aim to research technical approaches, e.g.
 - Evaluate implementation technologies, make build-or-by decisions
 - Evaluate a user story’s performance or load impact
 - Gain confidence with an approach before committing to an estimate
- **Functional spikes** aim to understand how users will work with the system, e.g.
 - Prototype interface through mock-ups, wireframes, storyboards etc.
 - Get feedback from stakeholders

Dealing With Uncertainty in Spikes

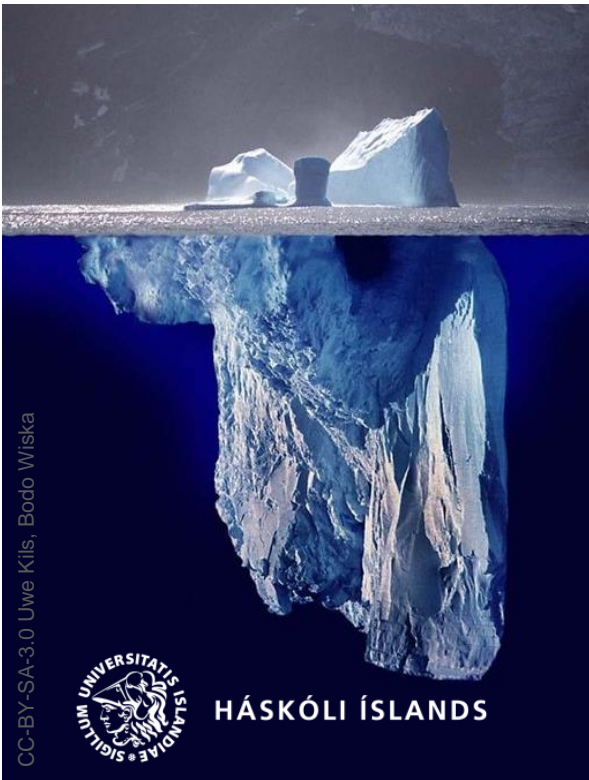
- Spikes do not directly deliver user value
 - Use sparingly and with caution!
- Guidelines:
 - Spikes are treated like other backlog items, i.e. estimated and sized to fit in an iteration
 - Spikes do not produce working code, but information, e.g. a prototype or basis for decision
 - Spike results should still be concrete and demonstrable (helps to justify effort to stakeholders)
 - Spikes should be the exception, not the rule
 - Each user story has inherent uncertainty, and will therefore involve some spike-like research
 - Explicit spike backlog items should be reserved for larger, more critical uncertainties
 - Spike should not be planned in the same iteration as the user stories relying on its outcome
 - Otherwise, the planning of the user story cannot benefit from the knowledge gained in the spike

User Stories Are Just the Tip of the Iceberg

- Even after refining user stories, you may find they still feel quite open-ended.
 - This is by design, since agile approaches deliberately eschew detailed requirements specs.
- You still need a more precise idea of user stories' intention to implement them.
 - The required knowledge can take many shapes:

- **User story**

- visible anchor of customer expectation, team commitment and requirements knowledge
-
- Knowledge from conversations with stakeholders
 - Attachments (e.g. examples, models, prototypes, use cases, data etc.)
 - Acceptance criteria (conditions of satisfaction placed on system)
 - Tasks (steps to be taken to ensure that acceptance criteria are satisfied)



Acceptance Criteria

- Acceptance criteria are conditions of satisfaction being placed on the system.
 - “Based on which criteria can the customer and supplier agree that this user story has been implemented satisfactorily?”
- Acceptance criteria are not functional or unit tests, but are a first step towards making user stories evaluable against concrete criteria.
 - In effect, they are yet one more detailed level of the user story’s definition
 - Describing the solution in business terms
- Example user story:
 - As a consumer, I want to be able to see my daily energy usage so that I can lower my consumption and cost.
- Example acceptance criteria:
 - Read deciwatt meter data every 10 seconds and display on portal in 15-minute increments.
 - Read kilowatt meter for new data as available and display on the portal every hour.

Issues with User Stories

- User stories don't give developers a context to work from
 - When is the user doing this? What is the context of the operation? What is their larger goal?
- User stories don't give any sense of scope or completeness
 - How “deep” is a user story? What exceptions lurk inside? What's outside these spotlights?
- User stories don't provide a mechanism for looking ahead at upcoming work
 - Hard to foresee main and alternate scenarios; extensions usually recognized mid-iteration
- A list of backlog items is easy to comprehend, manage and prioritize, but inadequate to provide the context required to design larger systems.

Benefits of Use Cases (Yes, in Agile Projects!)

- A UC's main success scenario states exactly what the system will and will not do
 - Provides context for each requirement, and agreement on scope between biz and tech side
- A UC's alternate scenarios answer the many tricky, detailed business questions that no business person initially thinks of but every developer soon runs into:
 - What are we supposed to do in this special case? What if that goes wrong in the process?
- A UC's list of goals provides a summary of the system's benefits & added value
 - Helpful for initial prioritizing, team allocation and scheduling
- A UC's extension conditions highlight business and technical complexity drivers
 - These would otherwise surface while a requirement is worked on, and delay it unforeseeably
- Full set of UCs shows that developers have considered each user's needs, each business goal and variant.
 - Gives customer and team more confidence that estimates and schedules are realistic.

Capturing Use Cases

1. Identify and describe the actors

- Who uses the system?
- Who gets information from this system?
- Who provides information to the system?
- Where is the system used?
- Who supports and maintains the system?
- What other systems or devices use this system?

2. Identify the use cases

- What will the actor use the system for?
- Will the actor create, store, change, remove, or read data in the system?
- Will the actor need to inform the system about external events or changes?
- Will the actor need to be informed about certain occurrences in the system?

Capturing Use Cases

3. Identify the actor and use case relationships

- Which actor initiates the use case?
- Which actors are participating in the use case?

4. Outline the flow of the use cases

- Basic flow (main success scenario)
 - What actor's event starts the use case?
 - How does the use case end?
 - How does the use case repeat some behavior?
- Alternate flows (alternate scenarios)
 - What else can the actor do?
 - How will the actor react to optional situations?
 - What variants might happen?
 - What exceptions to the usual behavior may occur?

Capturing Use Cases

5. Refine the use cases

- What are alternate flows, including unusual exception conditions?
- What preconditions must be satisfied before a use case can start?
- What exit conditions will the use case leave behind?
 - Success guarantee (state after successful execution)
 - Minimum guarantee (state after unsuccessful execution)

Use Cases in an Agile Context

- Use cases can be useful inside and outside user stories
 - **“inside”**: A use case is attached to a particular user story, in order to specify how exactly the interaction demanded by the story shall play out
 - **“outside”**: A use case describes a complex interactive feature, whose realization will be broken into several user stories for project planning purposes
- Guidelines: When applying use cases in agile projects,
 - Keep them lightweight, i.e. no design details, GUI specs etc.
 - Don't treat them like fixed requirements
 - Like user stories, they just state intended system behavior
 - Model them informally, and don't worry about maintaining them
 - They are tools to help you understand the requirements, not to document them
- Use cases are a proven tool to think through complex system requirements
 - Highly recommended to use them independently of the chosen software process model

Tasks

- Once a backlog item (user story or other work item) is sufficiently understood in terms of the *requirements* it poses...
- ...it's time to consider *what needs to be done* to implement those requirements
 - i.e. which concrete activities team members need to perform to realize the backlog item
 - Def.: A task is a small unit of work that is necessary for the completion of a backlog item.
- This also is a form of refinement of the user story
 - but while *acceptance criteria* refine the user story in *business terms* (product/user view), *tasks* refine it in *technical, operational terms* (project/developer view)
- Tasks help the team to understand what needs to be done
 - most precise basis for estimating efforts
 - interface between analysis/design and construction/controlling work in each iteration

Example Tasks

- **User story:**
 - Select photo for upload
- **Tasks for story:**
 - Define acceptance test
 - Code story
 - Code acceptance test
 - Get test to pass
 - Document feature in user help

Summary: Agile Requirements Engineering

- Start by brainstorming on features
- Take a step back and consider needs, potentially epics
- Take a closer look and consider requirements (typ. in user story format)
- Schedule spikes to resolve uncertainties
- Refine user stories
 - Discuss with customer
 - Elaborate and/or split
 - Work out use cases to understand particularly complex scenarios
- Define acceptance criteria
- Define tasks to be performed in order to satisfy acceptance criteria
- Prioritize requirements, estimate efforts, plan iterations (*→ HBV401G, Lect. 4 & 5*)

Reflection: Requirements Engineering in Agile vs. Plan-driven Projects

- **Similarities** between agile and plan-driven requirements engineering:
 - Expect to spend just as much time on requirements work in either process
 - This is natural: Impossible to circumvent the need and effort of understanding the business domain well enough to implement the system correctly
 - Requires a lot of learning, and critical reflection of one's own knowledge and assumptions
 - Requires close collaboration with the customer
- The **differences** are that agile requirements engineering...
 - ...sees written requirements as starting points of clarification, rather than endpoints
 - ...permeates the whole project, rather than occurring in occasional planned phases
 - ...takes many forms (user stories, discussions), rather than materializing in one document
- **Refactoring is not an efficient alternative to diligent requirements work!**
 - Refactor mostly to integrate new features into existing “minimum viable” code (pragmatic)...
 - ...but don't keep refactoring things just because you didn't think them through (expensive)

Assignment 1: Project Plan and Requirements

Note: Slightly updated from draft shown in first week.

- By **Sun 27 Jan**, submit a **project outline** in Uгла, containing:
 - Project vision (in form of a RUP Vision document, Data Sheet or Press Release)
 - Product backlog (user stories with effort estimates and priorities)
 - Project schedule (dates for sprints, milestones, assignments 2 and 3)
- On **Thu 31 Jan**, present and **explain** your project outline to your tutor:
 - What influenced your choice of scope? What considerations shaped your planned schedule?
- **Grading criteria:**
 - Project vision is clear and plausible (25% of this assignment's grade)
 - Product backlog describes requirements clearly and with realistic scope (50%)
 - Project schedule is clear, realistic, and specifies dates for assignments 2 and 3 (25%)