Hugbúnaðarverkefni 2 / Software Project 2

2. Agile Requirements

HBV601G – Spring 2020

Matthias Book

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

Requirements in Agile Projects

see also:

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

Recap: The Nature of Software Development

“Because software is embodied knowledge,
and that knowledge is initially dispersed, tacit, latent, and incomplete,
software development is a social learning process.”

Howard Baetjer, Jr.: Software as Capital. IEEE Computer Society Press, 1998

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 rely on the customers to “just 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 req’s 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 late 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 2)*

- 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

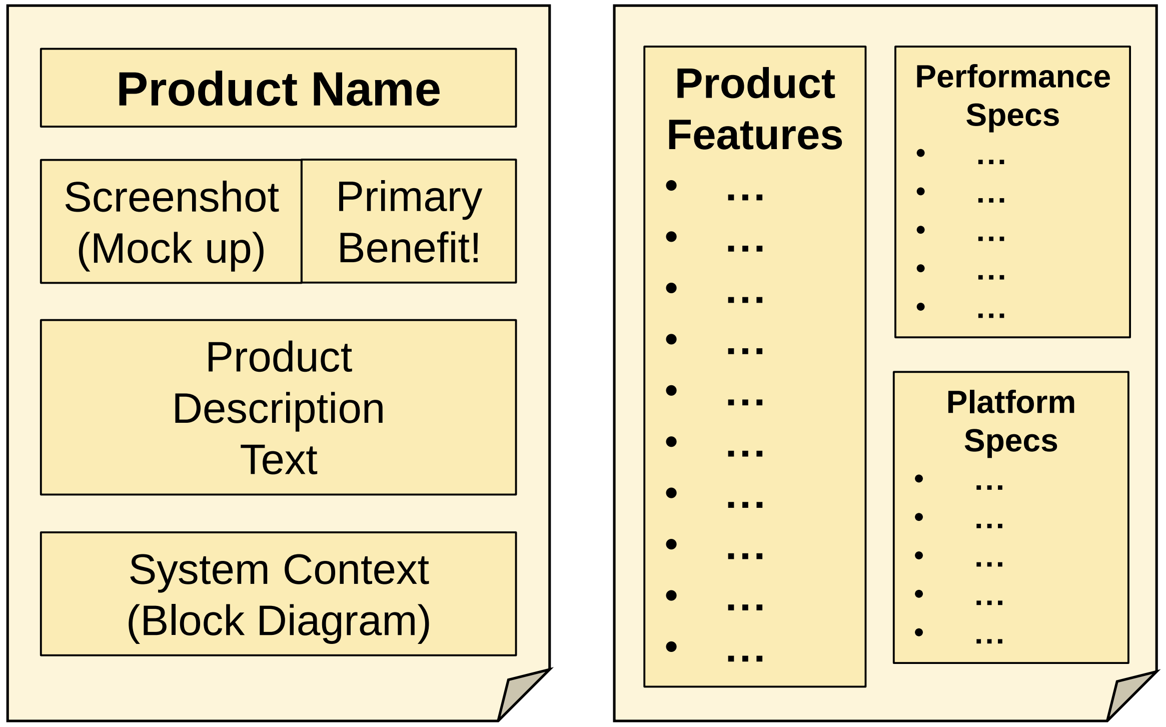


Figure adapted from D. Leffingwell, Agile Software Requirements. Pearson, 2011

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

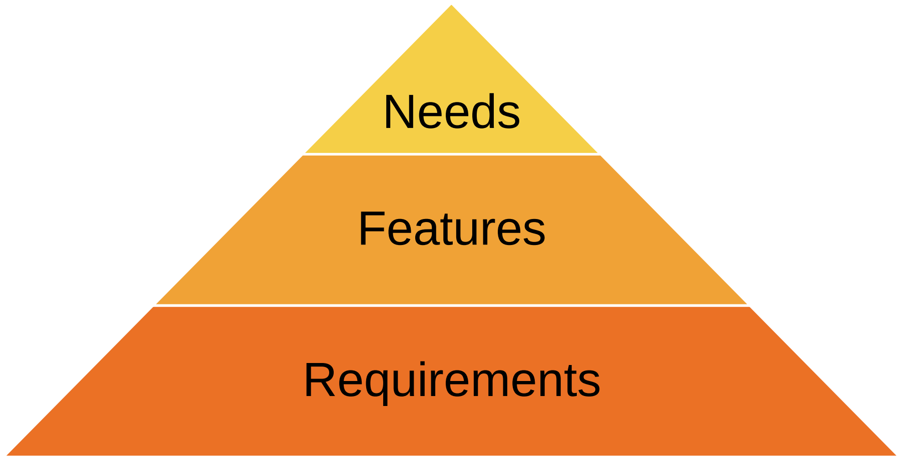


Figure adapted from D. Leffingwell, Agile Software Requirements. Pearson, 2011

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.

Self-Check Quiz: Types of Requirements

HBV401G, Lecture 5

- **Indicate if the following are (E)pics, (F)eatures, (N)eeds, (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

Self-Check Quiz: Types of Requirements (Solution)

- **Indicate if the following are (E)pics, (F)eatures, (N)eeds, (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. (R)

b) Define acceptance test (T)

c) In addition to physical store, sell goods also in an online shop. (E)

d) Payment by credit card (F)

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 (N)

f) Cancel an order (F)

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, Lectures 4 & 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 phrased 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.	*Main scenario:* Notify companies of request and handle transfer: […] *Alternative scenario:* 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.	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. Functional spike: Prototype a histogram in the web portal and get some user feedback on chart style and adjustment parameters.

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

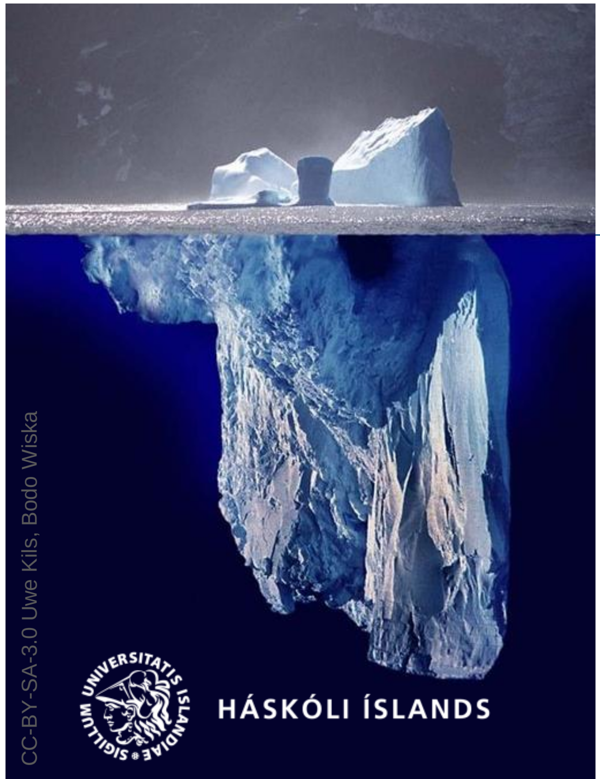
- Spikes do not directly deliver *user* value

➢ 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 Requirements 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.

adapted from A. Cockburn:

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

- A UC’s <u>main success scenario</u> 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 <u>alternate scenarios</u> 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 <u>goals</u> provides a summary of the system’s benefits & added value
  - Helpful for initial prioritizing, team allocation and scheduling
- A UC’s <u>extension conditions</u> 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.

adapted from A. Cockburn:
A user story is to a use case as a gazelle is to a gazebo

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 too much 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

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 explicitly 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)

Recap: Assignment 1: Project Plan and Requirements

Start on #1&2 this week

- By **Sun 2 Feb**, submit a **project outline** in Ugla, containing:
  1. Project vision (in form of a RUP Vision document, Data Sheet or Press Release)
  2. Product backlog (prioritized user stories)
  3. User Story estimates (three-point estimates of expected cases, using basic PERT formula)
  4. Project schedule (dates for sprints, milestones, assignments 2 and 3)

Start on #3&4 next week

- On **Thu 6 Feb**, present and **explain** your project outline to your tutor:
  - What considerations influenced your choice of scope, estimates and schedule?

- **Grading criteria:**
  - Project vision is clear and plausible (15% of this assignment's grade)
  - Product backlog describes requirements clearly and with realistic scope (40%)
  - Project estimates calculated using appropriate formulae (35%)
  - Project schedule is clear, realistic, and specifies dates for assignments 2 and 3 (10%)

Hugbúnaðarverkefni 2 / Software Project 2

3. Software Estimation

HBV601G – Spring 2020

Matthias Book

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

In-Class Quiz Prep

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

ID: _____@hi.is   Date: _________

1. a) _____   b) _____

2. a) _____ - _____ b) _____ - _____
   c) _____ - _____ d) _____ - _____
   e) _____ - _____

- 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

Software Estimation

see also:

- McConnell: Software Estimation, Ch. 1, 3-5, 7-13
- Leffingwell: Agile Software Requirements, Ch. 8

Accurate Estimates Are Based on Precise Requirements

A seemingly trivial requirement:

Users must enter a valid phone number with their order.

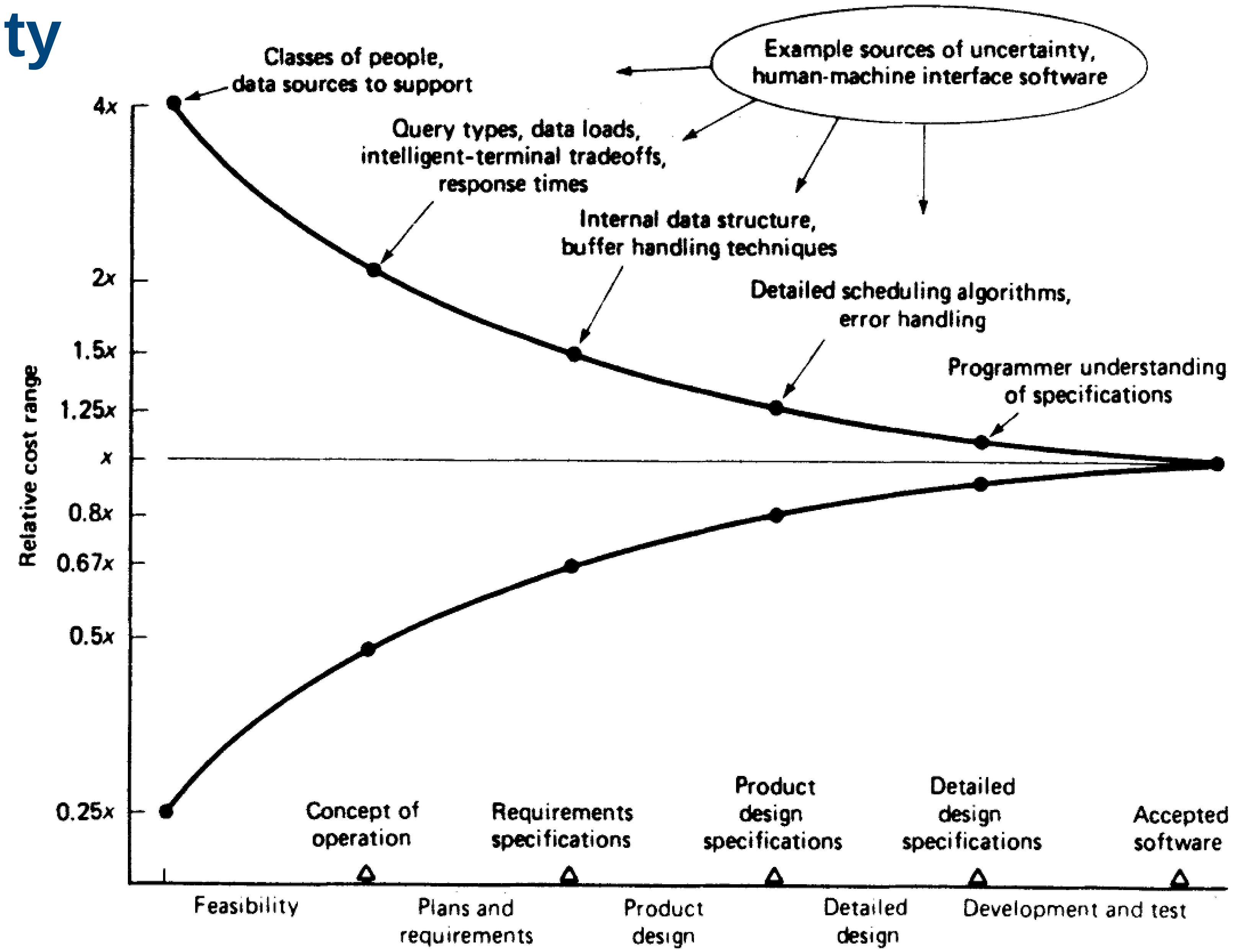
But we need to understand:

- This is not just about *entering*, but about *validating* a number using some “phone number checker (PNC)”
- If the customer wants a PNC, will he want the version we can build in 2 hours, 2 days, or 2 weeks?
- If we implement the cheap version of the PNC, will the customer later want the expensive version after all?
- Can we use an off-the-shelf PNC, or are there design constraints that require us to develop our own?
- Do the PNC and the Address Checker need to interact? How long will it take to integrate them?
- How will the PNC be designed?
- How long will it take to code the PNC?
- What will the quality level of the PNC be?
- How long will it take to debug and correct mistakes made in the PNC implementation?

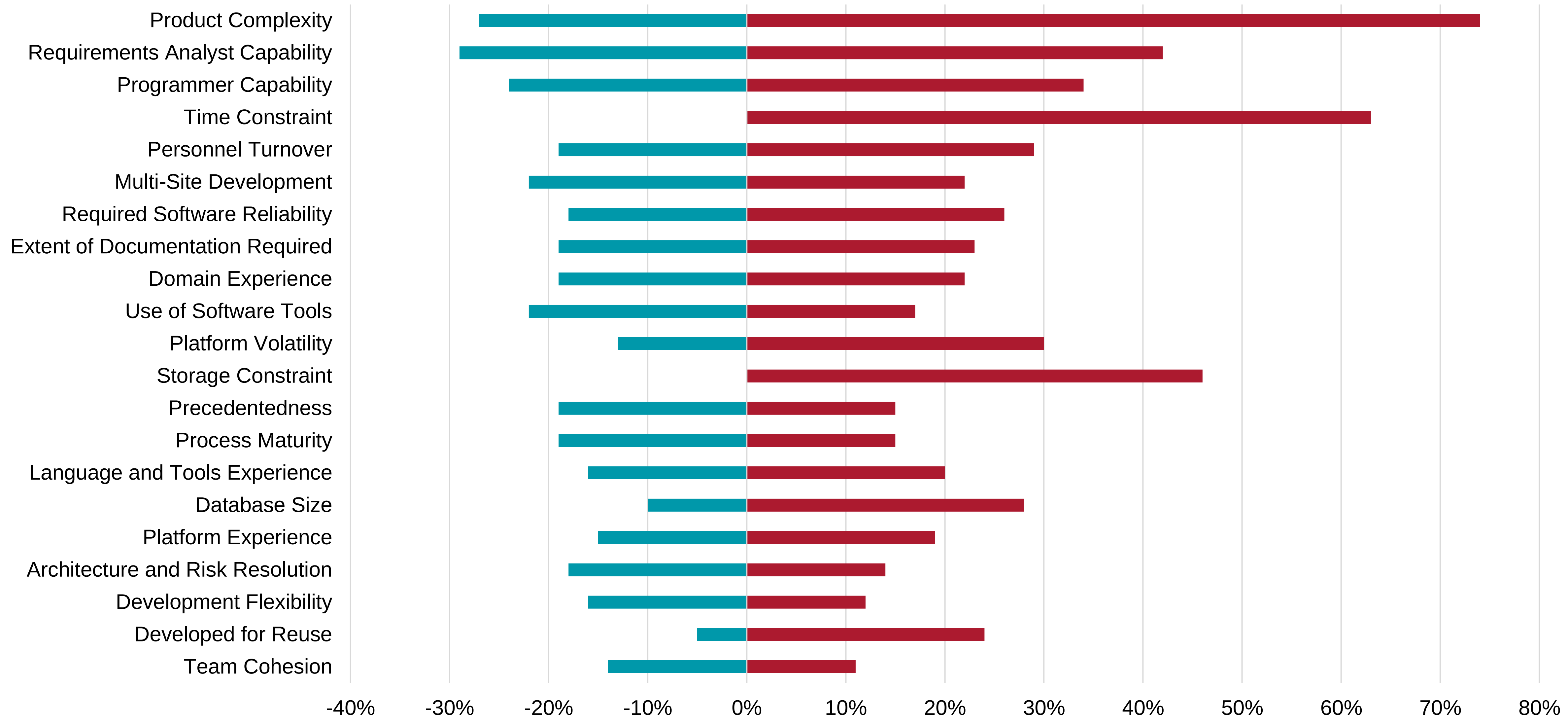
Each of these can introduce a factor 10 of difference in complexity, quality, time etc.

Cone of Uncertainty

- A measure for the maximum possible estimation accuracy over the course of a project
  - Your cone may be wider
  - i.e. your accuracy may be worse!
- The cone does not narrow by itself
  - Continuous active work is required to eliminate sources of uncertainty in the project.



Impact Bandwidth of Factors Affecting Project Effort

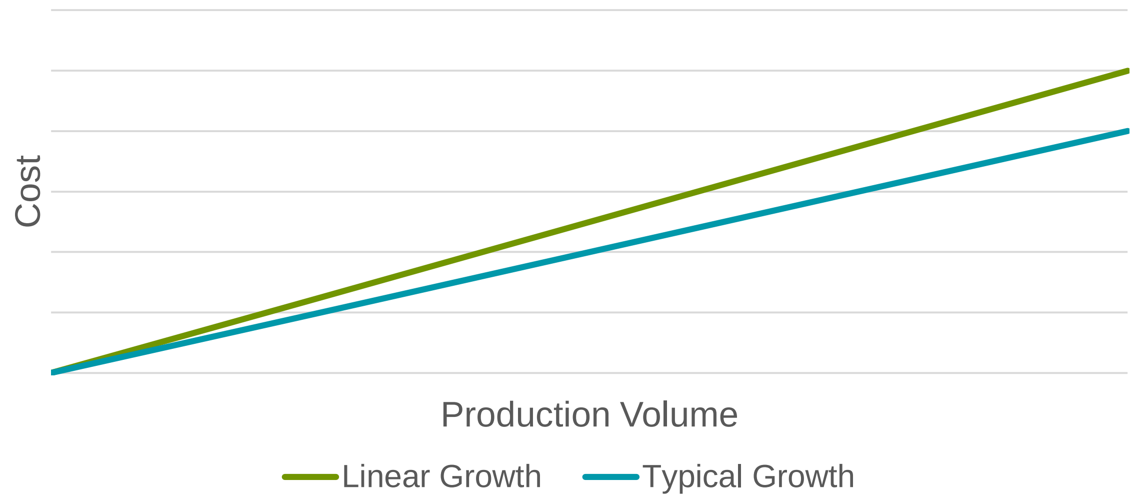


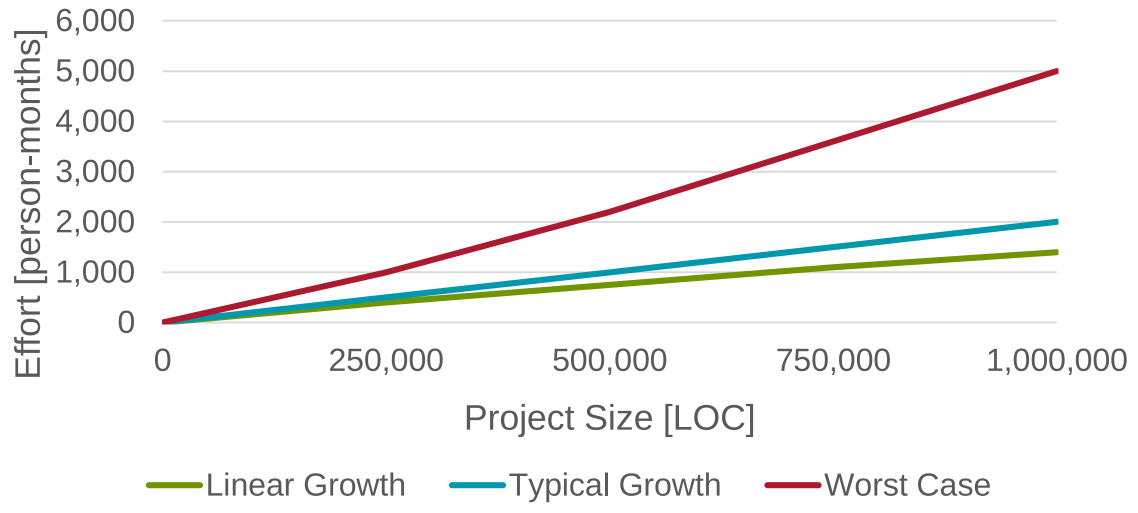
Diseconomies of Scale

- Many manufacturing domains can benefit from **economies of scale:** “The more units we produce, the smaller the cost of each unit.”

- In software engineering, we have to deal with a **diseconomy of scale:** “The larger a system we build, the higher the cost of building each part.”

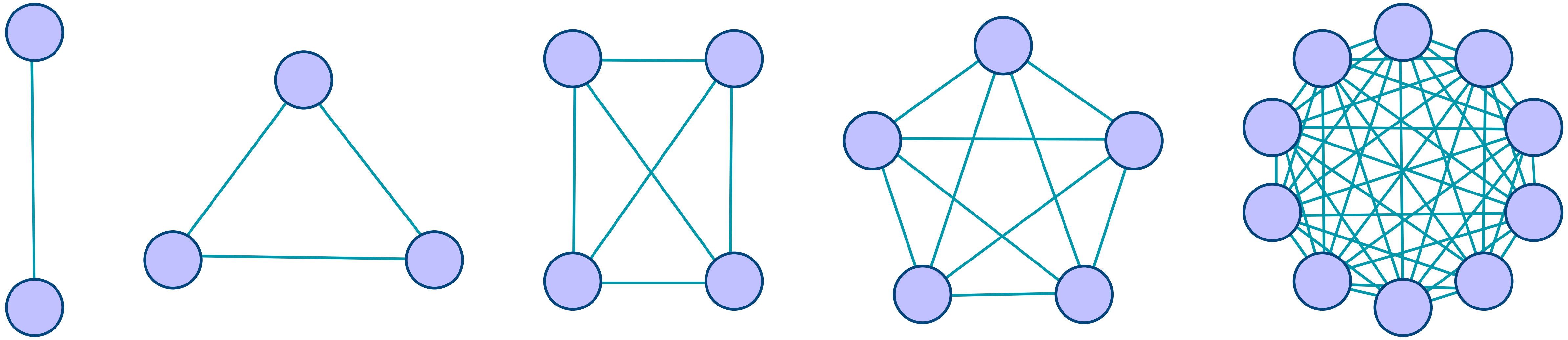
- Effort does not scale linearly with project size, but exponentially.





Diseconomies of Scale

- A system that is 10 times as large requires *more* than 10 times as much effort.
  - A linear increase in components (or a linear increase in team size) results in an exponential increase in **relationships** between components (or people).
  - Of course, efficient component design and encapsulation (or effective team structures) can reduce the number of actual relationships considerably,
  - but the cognitive complexity that the team has to deal with remains high.



Difficulties in Software Estimation

- **Caution when interpreting management requests**
  - Determine whether you are expected to estimate, or figure out how to hit a target

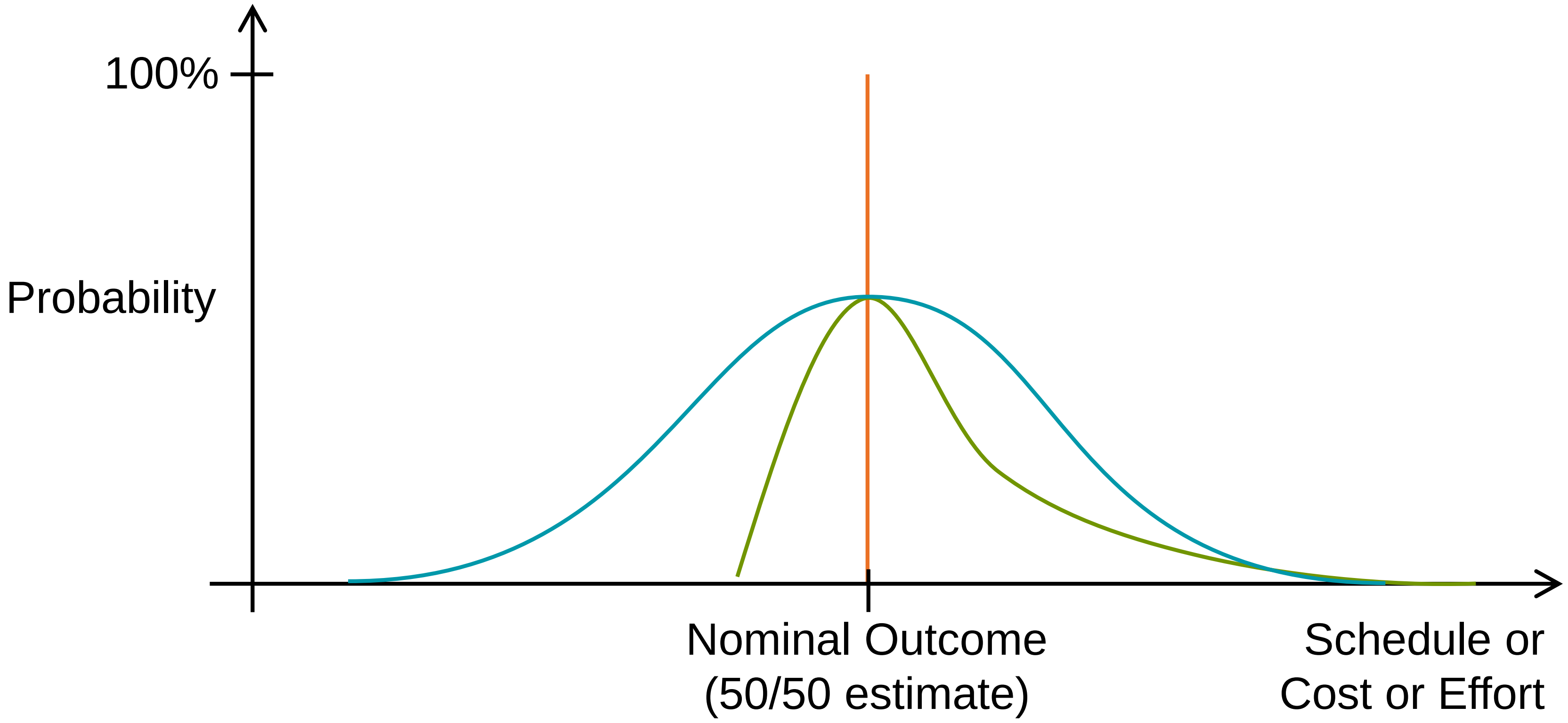
- **Caution when preparing estimates**
  - Hard to quantify the subject
  - Lots of confounding factors
  - Temptation to tweak the estimate

- **Caution when letting management interpret your estimates**
  - Your estimates may be understood as precise commitments

Estimates as Probability Statements

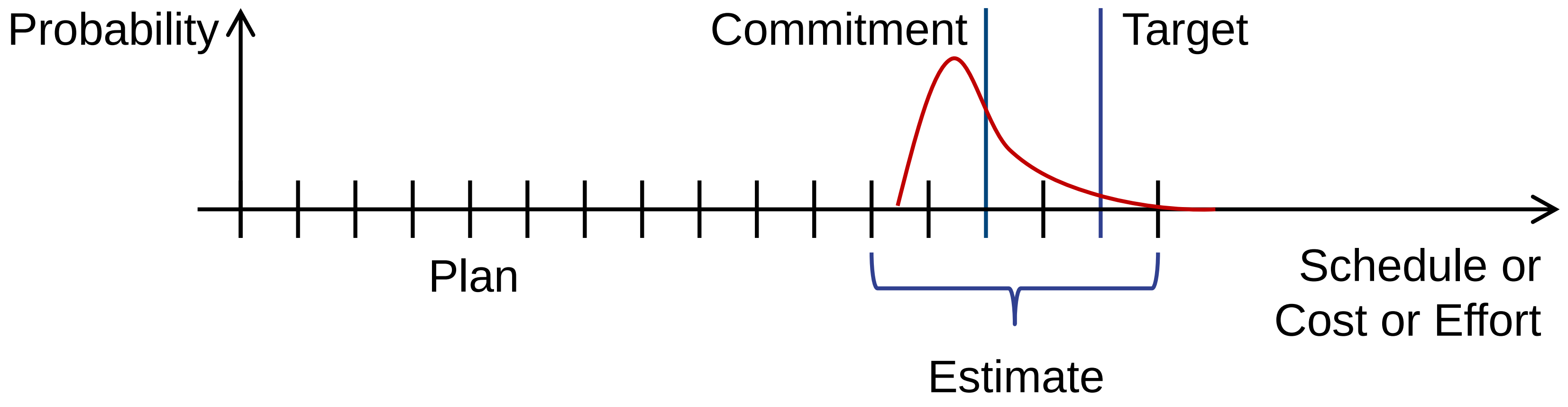
- ~~Single point estimate: Assumes that estimate will accurately predict result~~
- ~~Bell curve: Disregards limits of how efficiently a project team can complete work~~

✓ Realistic curve: There's a limit to how *well* a project can go, but not how *bad*



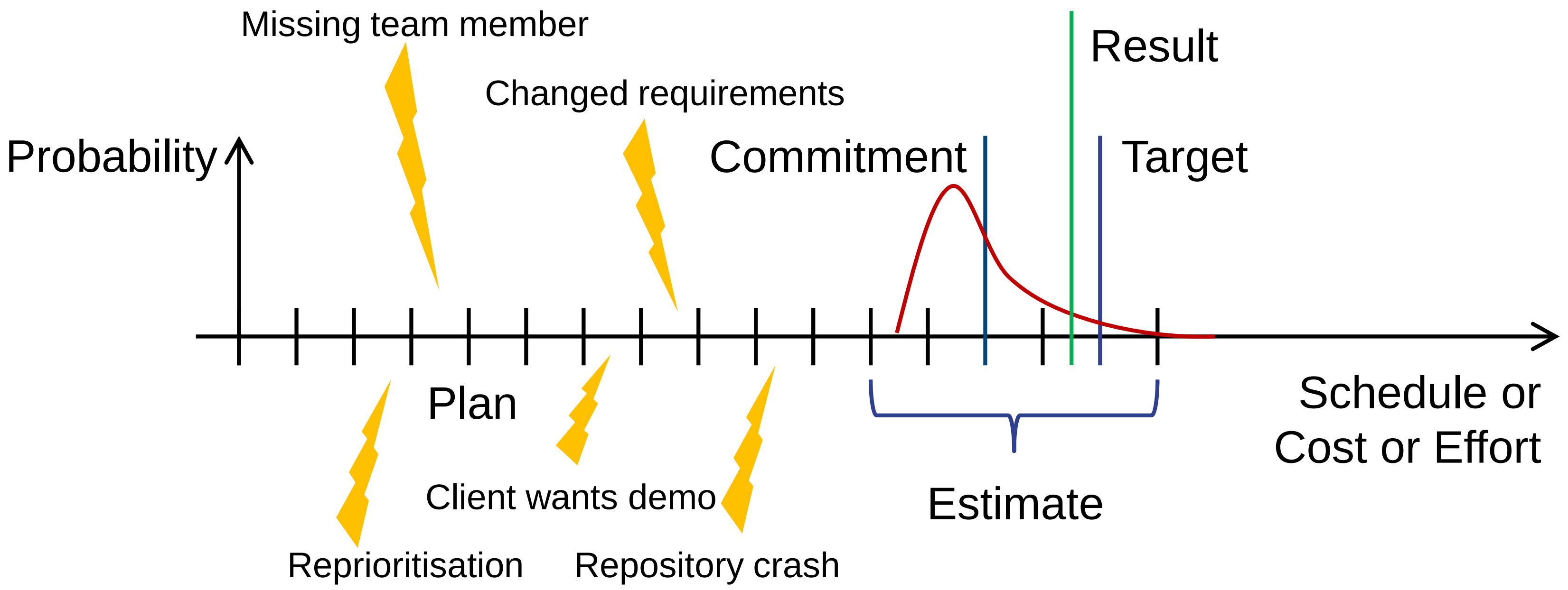
Terminology

- **Target:** fixed value (deadline, budget etc.) determined by external factors
- **Estimate:** preliminary prediction of a future value/interval
- **Commitment:** Intention of reaching a certain future value
- **Plan:** Agreement of steps to achieve the commitment



Life Is What Happens While You Are Making Other Plans

- Some project events make (the foundations of) prior estimates obsolete.



The Purpose of Software Estimation

- The aim of effort estimates is not to predict the project result, but to **judge whether the project target is sufficiently realistic** to be **achievable through corresponding project management.**
  - Estimates do not need to be extremely precise, as long as they are **useful**.
- Experience: If the initial target and the initial estimate are within ~20% of each other, the project manager will have enough maneuvering room (in terms of features, schedule, staffing etc.) to meet the project's business goals.
- If the gap between target and estimate is larger, it is very unlikely the project can be completed successfully.
  - ➢ Bring target into better alignment with reality before the project has a chance of success.
  - **RESIST TEMPTATION / PRESSURE** to align estimate better with target!
    - Your first estimate is rough but honest.
    - Revisions not based on additional data are just kidding yourself and endangering your project.

Estimation Approaches Discussed in This Class

A. Projection from counts
   1. Concrete items
   2. Function points

B. Individual expert judgment

C. Group expert judgment
   1. Wideband Delphi
   2. Planning Poker

D. Qualitative estimation

A_1) Projection from Counts

1. Find a suitable type of item to count
   - Highly correlated with size of software you're estimating
   - Available early in development cycle
   - Statistically meaningful number of items available (>20)
     - If we are talking about less, the individual differences between these items become too dominant
   - Countable with minimal effort
   - Historical averages based on comparable parameters(!) available
     - Similar product, similar team, similar technology, similar complexity…

2. Count items of interest

3. Calculate effort from item counts, based on average effort per item

Countable Items for Projecting Estimates

- Marketing requirements
- Engineering requirements
- Features
- Use cases
- User stories
- Function points
- Change requests
- Web pages
- Reports
- Dialog boxes

- Database tables
- Classes
- Defects found
- Configuration settings
- Lines of code already written
- Test cases already written
- …

- **Projection:** Multiply counted items with average effort per item observed in historical projects of similar kind

In-Class Quiz #1: Projecting Estimates from Counts

a) Counting defects late in project

- For the last 250 defects we fixed, we needed 2 hours per defect on average.
- 400 defects are currently open in our project.

- How many hours will it approximately take us to fix the open defects?

b) Counting web pages

- So far, each dynamic web page in the project cost us a total of 40 hours to design, code and test, on average.
- We have 10 web pages left to build.

- How many hours will it approximately take us to finish this?

A_2) Function Points (FP)

- Synthetic measure of software size assigned to counts of various types of items:
- **External inputs**
  - Screens, forms, dialog boxes, control signals through which an end user or other system adds, deletes, or changes our system’s data
- **External outputs**
  - Screens, reports, graphs, control signals that our system generates for use by an end user or other system.
- **Internal logical files**
  - Major logical groups of end-user data or control information completely controlled by our system, e.g. a single flat file or a single table in a database
- **External interface files**
  - Files controlled by other systems with which our system interacts; this includes each major logical group of data or control information entering or leaving our system

Function Points

- Each system characteristic is classified by complexity (low / medium / high) and assigned an according number of function points:

System Characteristic	Low Complexity	Medium Complexity	High Complexity
External inputs	3	4	6
External outputs	4	5	7
Internal logical files	4	10	15
External interface files	5	7	10

- FPs are an abstract, relative measure of complexity, not absolute size/time/effort
- Formulas exist to translate FPs into lines of code, based on industry experience
  - e.g. in Java, 1 FP requires at least 40, typically 55, at most 80 lines of code
- More useful: Use organizational experience to translate FP to expected effort

Recommendation: Avoid in practice – use story points if you want an abstract metric

Lines of Code (LOC)

Traditional size measure found in many tools and textbooks

Advantages

- Data easily collected and evaluated by tools
- Lots of historical data exists
- Effort per line of code is roughly constant across languages
  - (*Effect* per line of code obviously is not!)
- Most convertible "currency" between projects
  - But make sure projects are comparable to draw meaningful conclusions!

Disadvantages

- LOC don't reflect
  - Diseconomies of scale in software development effort
  - Difference in programmer productivity
  - Difference in expressiveness of programming languages
- Using LOC for estimating non-coding work (requirements, design etc.) is counterintuitive
- Which Lines of Code to count?

Recommendation: Avoid in practice – mostly useful for academic purposes

Experiment: Estimation Confidence

Estimate a range in which you are 90% confident the actual value will be:

a) Latitude of San Francisco:	▪ ______ – ______ °N
b) Birth year of Fridtjof Nansen:	▪ ______ – ______
c) Area of Iceland:	▪ ______ – ______ km^2
d) Production cost of movie “Avatar”:	▪ ______ – ______ million US$
e) Registered students at HÍ:	▪ ______ – ______

Quiz inspired by “Software Estimation” by Steve McConnell (Microsoft Press, 2006). The original is © 2006 Steve McConnell.

Experiment: Estimation Confidence

How many of the following actual values fall inside your estimation range?

a) Latitude of San Francisco:	▪ 37°47’ N
b) Birth year of Fridtjof Nansen:	▪ 1861
c) Area of Iceland:	▪ 103.000 km^2
d) Production cost of movie “Avatar”:	▪ 237 million US$
e) Registered students at HÍ:	▪ 13.092 (as of 20 Oct 2019)

In range	Percentage
0 of 5	0%
1 of 5	20%
2 of 5	40%
3 of 5	60%
4 of 5	80%
5 of 5	100%

Quiz inspired by “Software Estimation” by Steve McConnell (Microsoft Press, 2006). The original is © 2006 Steve McConnell.

Lesson: Estimation Accuracy

- Remember you were asked to provide a range that you were *90% confident* to include the actual value.

- Reflect:
  - Did you subconsciously try to make the estimate “better” by tightening the range?
  - Were you still highly confident that your range included the actual value?

- Perceived estimation confidence is usually much higher than actual confidence
  - Be cautious with statements like “90% confidence” – usually your estimate is far less reliable
  - Choose your estimation range deliberately
  - Don’t make the range narrower than necessary
  - The estimation range should reflect your inconfidence

Estimation vs. Commitment

- In software projects, people (and possibly yourself) will assume that your estimates can be directly translated into commitments.

I estimate we'll be done in 6 to 8 months.
Great, let's take the average and plan to ship in 7 months.
…9 months later:
$#%$#%!!

Estimation vs. Commitment

- Your estimate is much less reliable than you think,
  but other people even think it's much more reliable!
  - Carefully manage expectations and be clear about your confidence

…9 months later:

B) Individual Expert Judgment

- Let an expert (typically: developer) come up with an estimate for each feature
  - Caution: subjective, possibly based on non-transferable experience / incorrect assumptions

- Avoid developer single-point estimates
  - Usually, these reflect the developer's subconscious *best case* assumption!
- Create three-point estimates: Best Case, Most Likely Case, Worst Case
  - This will stimulate developers to think about full range of possible outcomes for each feature

- Calculate expected case from three-point estimates for each feature:
  - ExpectedCase = (BestCase + 4 * MostLikelyCase + WorstCase) / 6 (basic PERT formula)
  - ExpectedCase = (BestCase + 3 * MostLikelyCase + 2 * WorstCase) / 6
    - (alternative formula if you feel your team's "most likely" estimates tend to be too optimistic)

Example:
Calculating Expected Cases

Weighted average, e.g.
ExpectedCase = (BestCase +
4 * MostLikelyCase + WorstCase) / 6

Feature	Best Case	Most Likely Case	Worst Case	Expected Case
1	1.25	1.5	2.0	1.54
2	1.5	1.75	2.5	1.83
3	2.0	2.25	3.0	2.33
4	0.75	1	2.0	1.13
5	0.5	0.75	1.25	0.79
6	0.25	0.5	0.5	0.46
7	1.5	2	2.5	2.00
8	1.0	1.25	1.5	1.25
9	0.5	0.75	1.0	0.75
10	1.25	1.5	2.0	1.54
Total				**13.62**

[Days to Complete]

Checklist for Individual Estimates

- Is what’s being estimated clearly defined?
- Does the estimate include all the kinds of work needed to complete the task?
- Does the estimate include all the functionality areas needed to complete the task?
- Is the estimate broken down into enough detail to expose hidden work?
- Did you look at documented facts from past work or estimate purely from memory?
- Is the estimate approved by the person who will actually do the work?
- Is the assumed productivity similar to what has been achieved on similar assignments?
- Does the estimate include a Best Case, Worst Case, and Most Likely Case?
- Is the Worst Case really the worst case? Does it need to be made even worse?
- Is the Expected Case computed appropriately from other cases?
- Are you aware of the assumptions that are underlying this estimate?
- Has the situation changed since the estimate was prepared?

Training Your Estimation Accuracy Over Time

To improve estimation accuracy:

- Compare actual results to estimated results
  - Calculate Magnitude of Relative Error (MRE) for each feature estimate:
    MRE = | (ActualResult – EstimatedResult) / ActualResult |
    - The average of your MREs per sprint should decrease over time
  - Check if actual results were in range between best case and worst case
    - The percentage of actual results that are in this range should increase over time
  - Check if 50% of expected cases were overrun and 50% of them were underrun
    - If not, your individual estimates tend to be too optimistic or too pessimistic
      - Work on improving your best, most-likely and worst-case estimation accuracy
- Try to understand
  - What went right and what went wrong
  - What you overlooked
  - How to avoid making those mistakes in the future

Example: Calculating Magnitude of Relative Error

Feature	Best Case	Most Likely Case	Worst Case	Expected Case	Actual Outcome	MRE	Between best and worst case
1	1.25	1.5	2.0	1.54	2	23%	Yes
2	1.5	1.75	2.5	1.83	2.5	27%	Yes
3	2.0	2.25	3.0	2.33	1.25	87%	No
4	0.75	1	2.0	1.13	1.5	25%	Yes
5	0.5	0.75	1.25	0.79	1	21%	Yes
6	0.25	0.5	0.5	0.46	0.5	8%	Yes
7	1.5	2	2.5	2.00	3	33%	No
8	1.0	1.25	1.5	1.25	1.5	17%	Yes
9	0.5	0.75	1.0	0.75	1	25%	Yes
10	1.25	1.5	2.0	1.54	2	23%	Yes
Total				**13.62**	**16.25**	**29% avg.**	**80% yes**

[Days to Complete]