Dr. Michael Eichberg
Software Engineering
Department of Computer Science
Technische Universität Darmstadt

Software Engineering

Software Process Models

TECHNISCHE
UNIVERSITÄT
DARMSTADT

The Software (Engineering) Process is the set of activities and associated results that produce a software product.

- *Requirements specification*
- Software specification
  Definition of the software to be produced and the constraints of its operation.
- Software development
  Design and implementation of the software.
- Software validation
  To ensure that the software does what the customer requires.
- Software evolution
  Adaptation and modification of the software to cope with changing customer and market requirements.

Fundamental Process Activities

Software (Engineering) Process Models are simplified and abstract descriptions of a software process that present one view of that process.

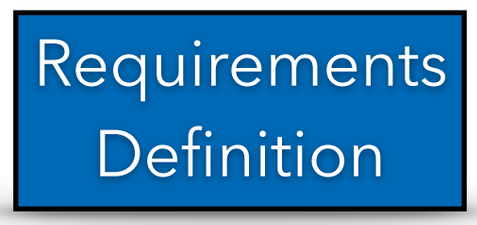
- Process models may include activities that are part of the software process, software products (e.g. architectural descriptions, source code, user documentation) and the roles of people involved in software engineering.
- Examples:
  - The waterfall model
  - Scrum
  - “V-Modell (XT)” (dt.)
  - eXtreme Programming
  - …

Large(r) projects may use different (multiple) software process models to develop different parts of the software.

The Waterfall Model

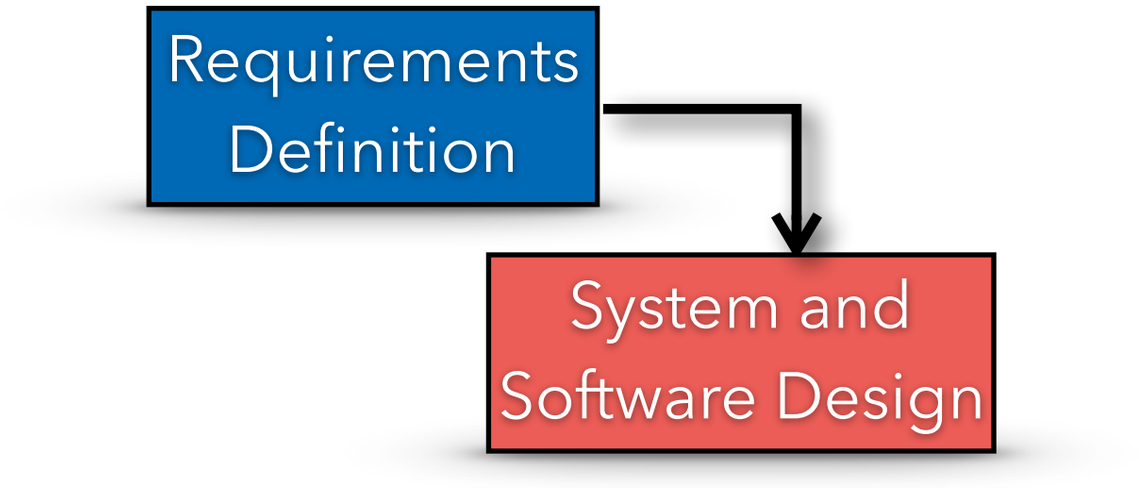
The Waterfall Model can be considered as a generic process model.

1. **Requirements analysis and definition**
   The requirements are established by consultation with system users. After that they are defined in detail and serve as the system specification.



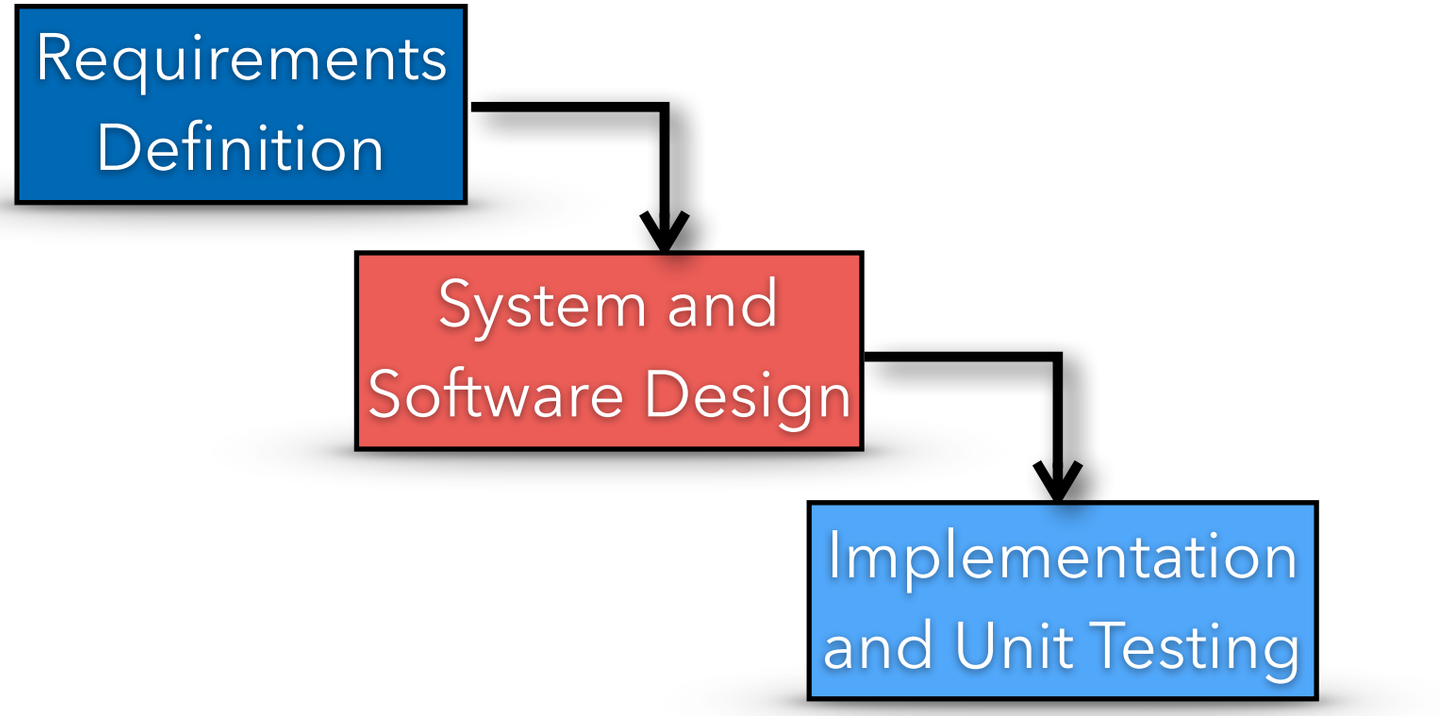
The Waterfall Model can be considered as a generic process model.

2. **System and Software design**
   The overall system architecture is defined. The fundamental software system abstractions and their abstractions are identified.



The Waterfall Model can be considered as a generic process model.

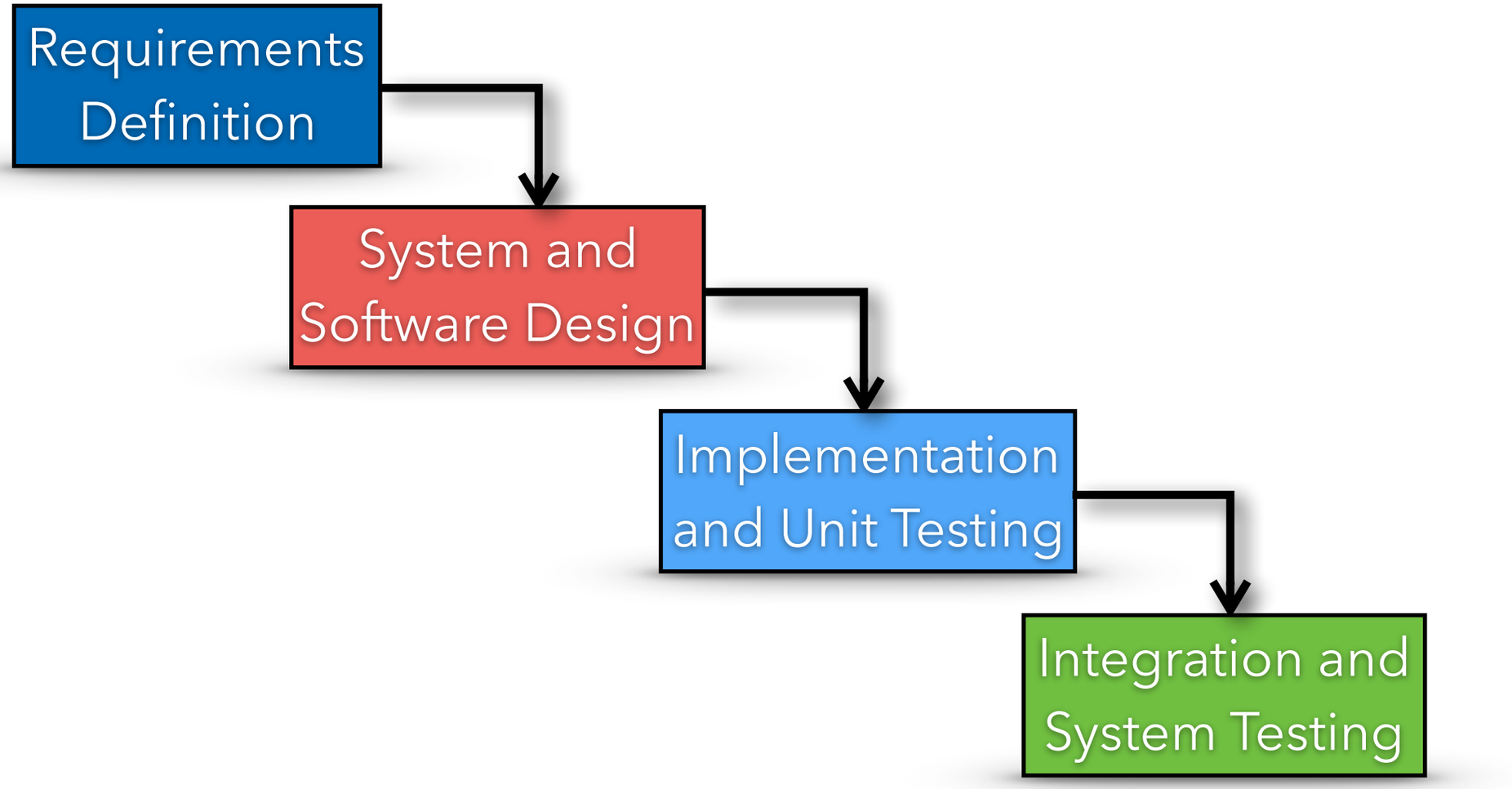
3. **Implementation and unit testing**
   The software design is realized as a set of program units; testing verifies that each unit meets its specification.



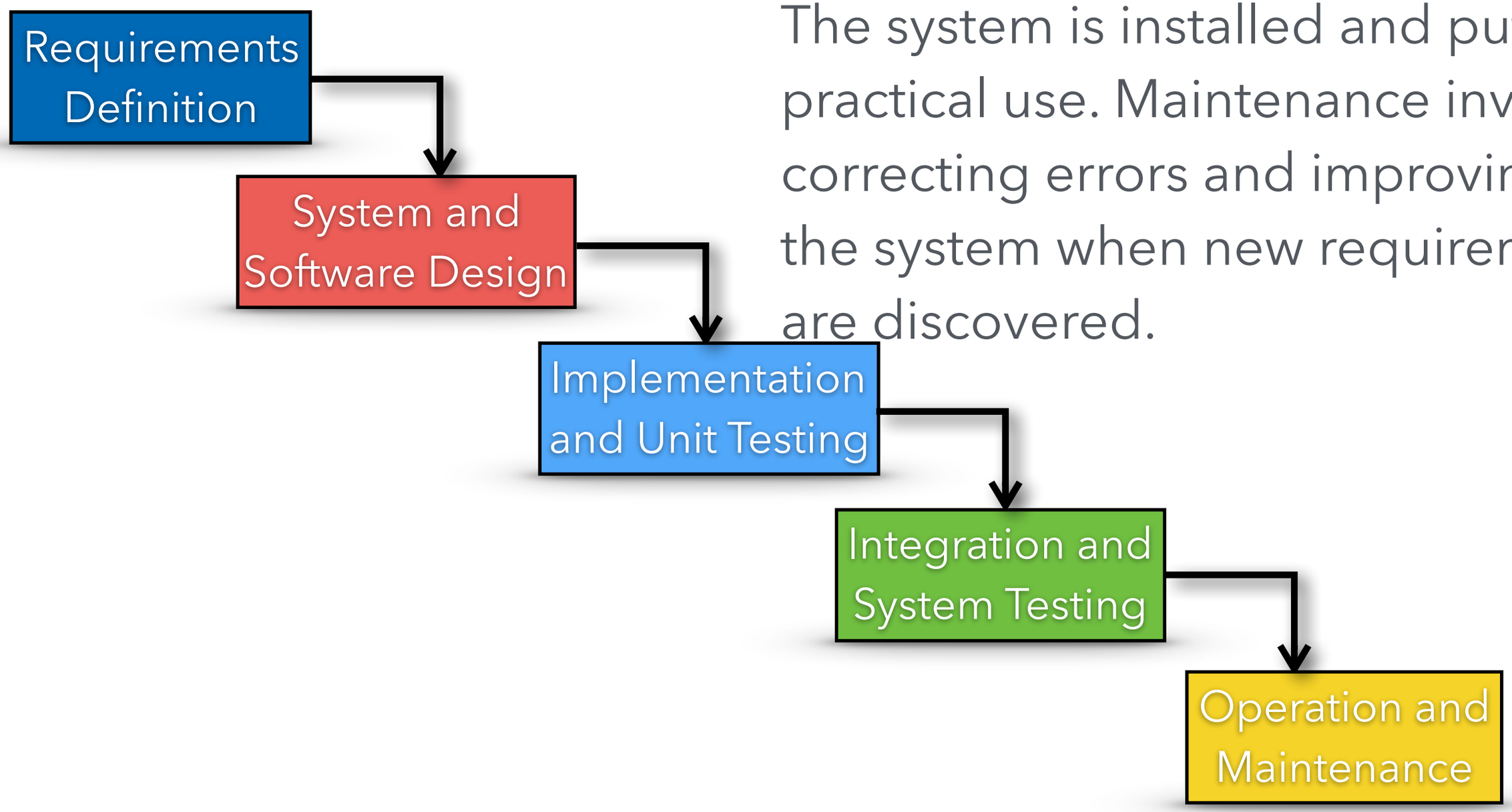
The Waterfall Model can be considered as a generic process model.

4. **Integration and system testing**

   Program units are integrated and tested as a complete system.

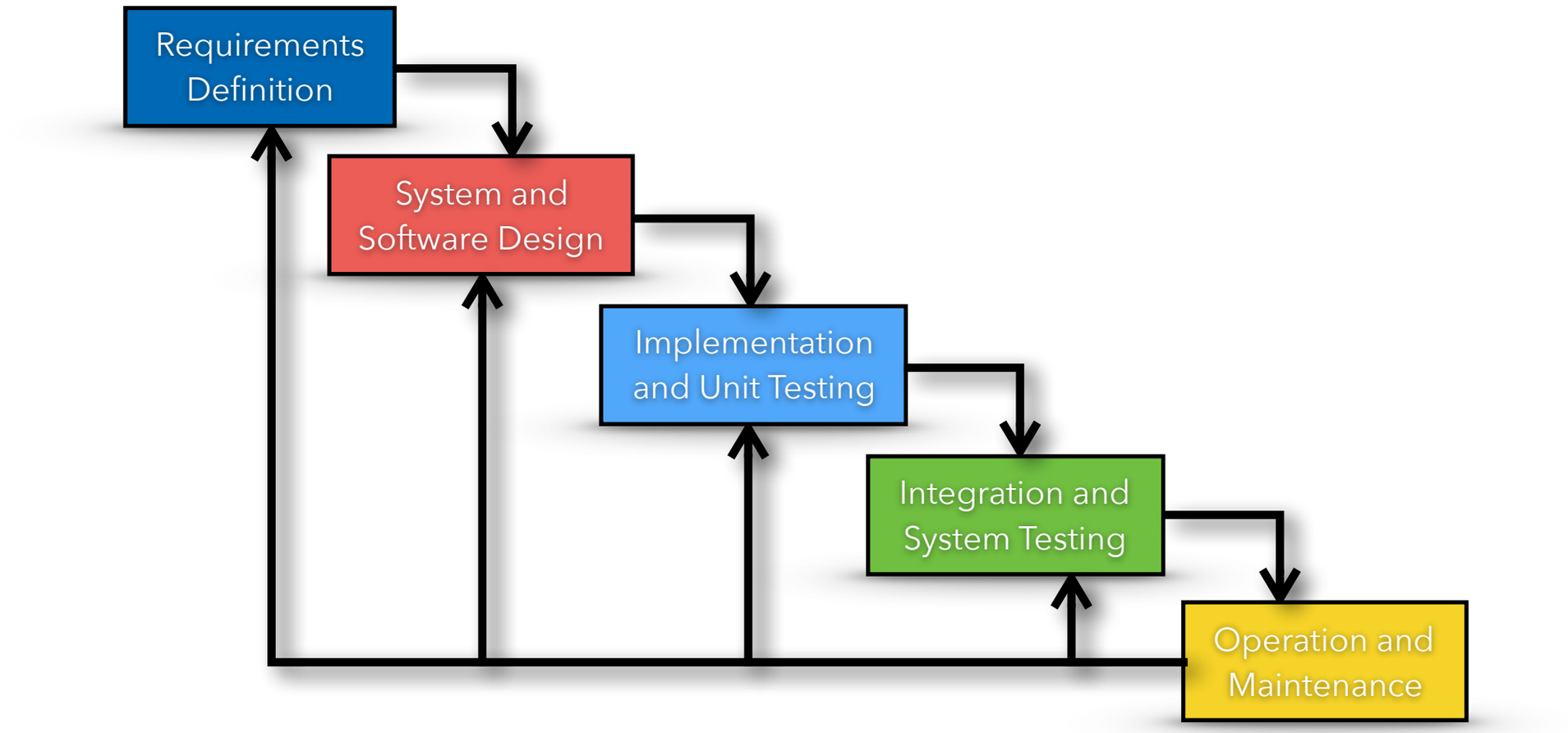


The Waterfall Model can be considered as a generic process model.



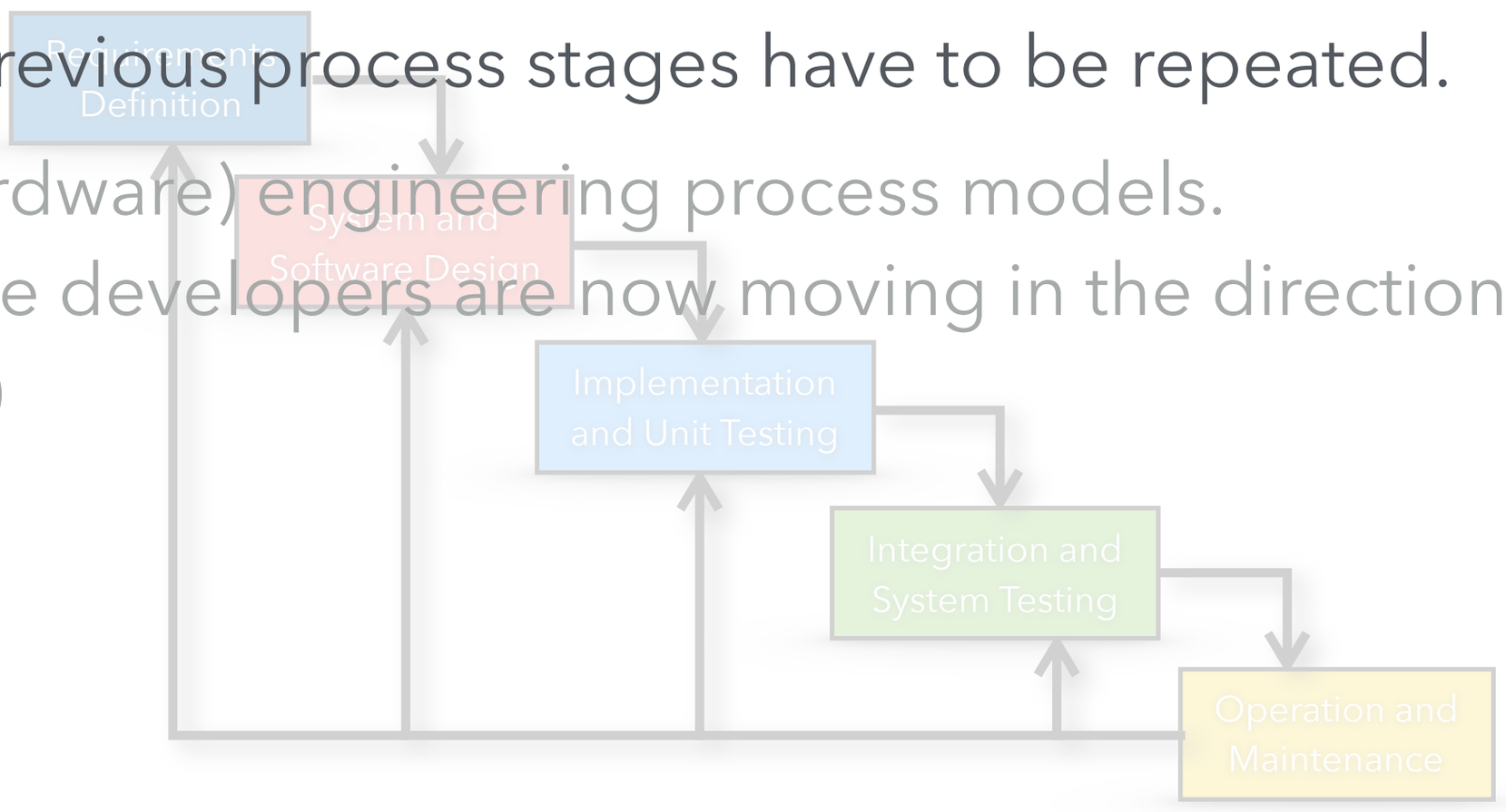
5. **Operation and Maintenance**
   The system is installed and put into practical use. Maintenance involves correcting errors and improving the system when new requirements are discovered.

The Waterfall Model can be considered as a generic process model.



Key Properties of the Waterfall Model

- The result of each phase is a set of artifacts that is approved.
- The following phase starts after the previous phase has finished. (In practice there might be some overlapping.)
- In case of errors previous process stages have to be repeated.
- Fits with other (hardware) engineering process models. (But even hardware developers are now moving in the direction of agile methods!)



Agile Development

- Agile Software Development - Principles, Patterns, and Practices; Robert C. Martin; 2003

Agile Methoden bauen auf iterativen Ansätzen auf.

Agile Development - Key Points

- **The goal is to develop software quickly, in the face of rapidly changing requirements**
- Originally conceived for small to mid-sized teams
- To achieve agility we need to ...
  - employ practices that provide the necessary discipline and feedback
  - employ design principles that keep “our” software flexible and maintainable
  - know the design patterns that have shown to balance those principles for specific problems

Using an agile method does not mean that the stakeholders will always get what they want. It simply means that they'll be able to control the team to get the most business value for the least cost.

Agile Development

- Manifesto

TECHNISCHE UNIVERSITÄT DARMSTADT

Manifesto for Agile Software Development

Individuals and interactions over process and tools.

The best tools will not help if the team doesn't work together.

Start small and grow if needed.

Manifesto for Agile Software Development

Working software over comprehensive documentation.

The structure of the system and the rationales for the design should be documented.

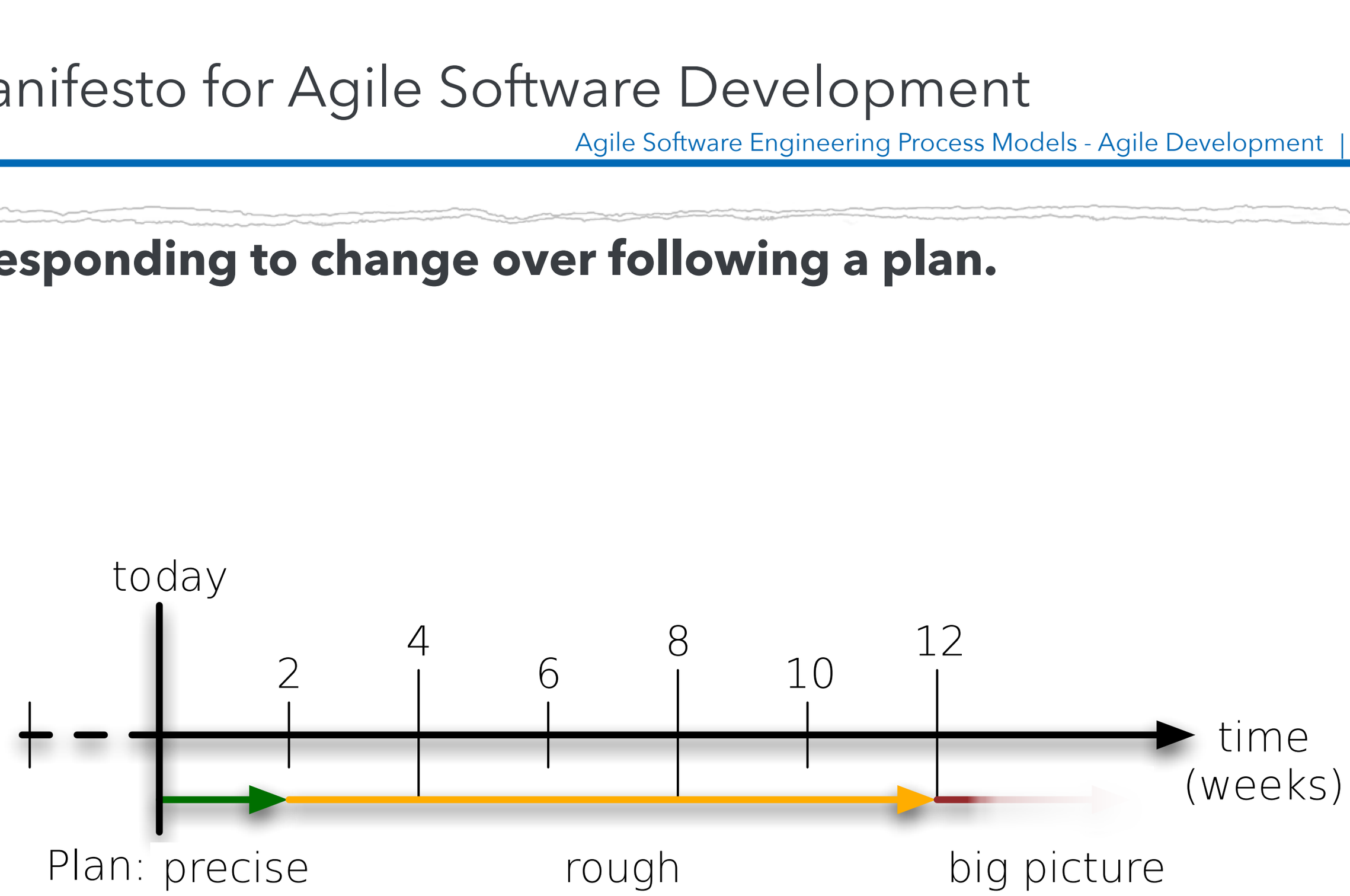
Manifesto for Agile Software Development

Customer collaboration over contract negotiation.

The contract should specify how the collaboration between the development team and the customer looks like.

Manifesto for Agile Software Development

Responding to change over following a plan.



Agile Development

- Principles

TECHNISCHE UNIVERSITÄT DARMSTADT

Principles of Agile Development

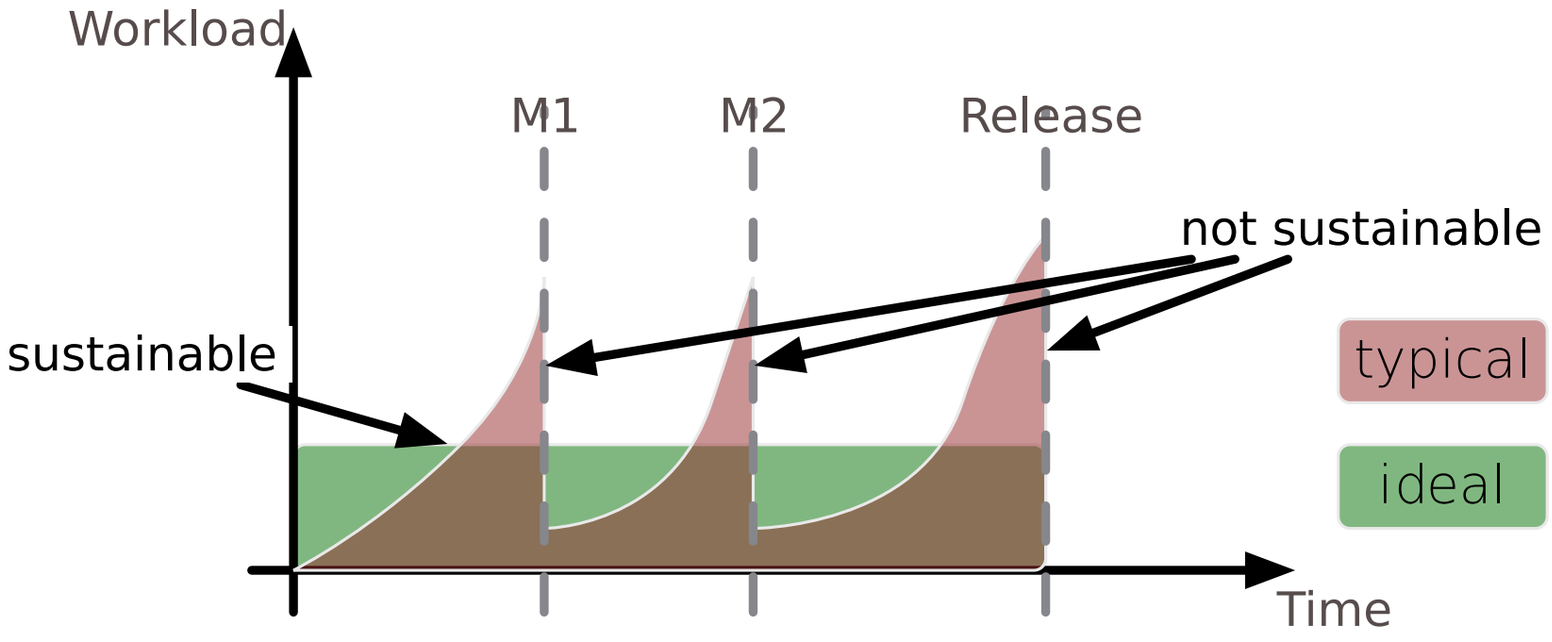
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Deliver working software frequently, from a couple of weeks to a couple of months, with a strong preference to the shorter timescale (e.g. every two weeks)
- Working software is the primary measure of progress
  If 30% of the functionality is implemented, 30% of the project is done.
- Continuous attention to technical excellence and good design enhances agility
- Simplicity - the art of maximizing the amount of work not done - is essential
- ...

Principles of Agile Development

- ...
- Welcome changing requirements, even late in development; agile processes harness change for the customer's competitive advantage
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly Process Improvement
- The best architectures, requirements, and designs emerge from self-organizing teams

Principles of Agile Development

- Business people and developers must work together daily throughout the project
- Build projects around motivated individuals; give them the environment and support they need, and trust them to get the job done
- Agile processes promote sustainable development; the sponsors, developers, and users should be able to maintain a constant pace indefinitely



Agile Processes

- SCRUM(~Project Management Method)
- (Agile) Unified Process
- Crystal
- Feature Driven Development
- Adaptive Software Development
- Extreme Programming
- ...

Unified Process

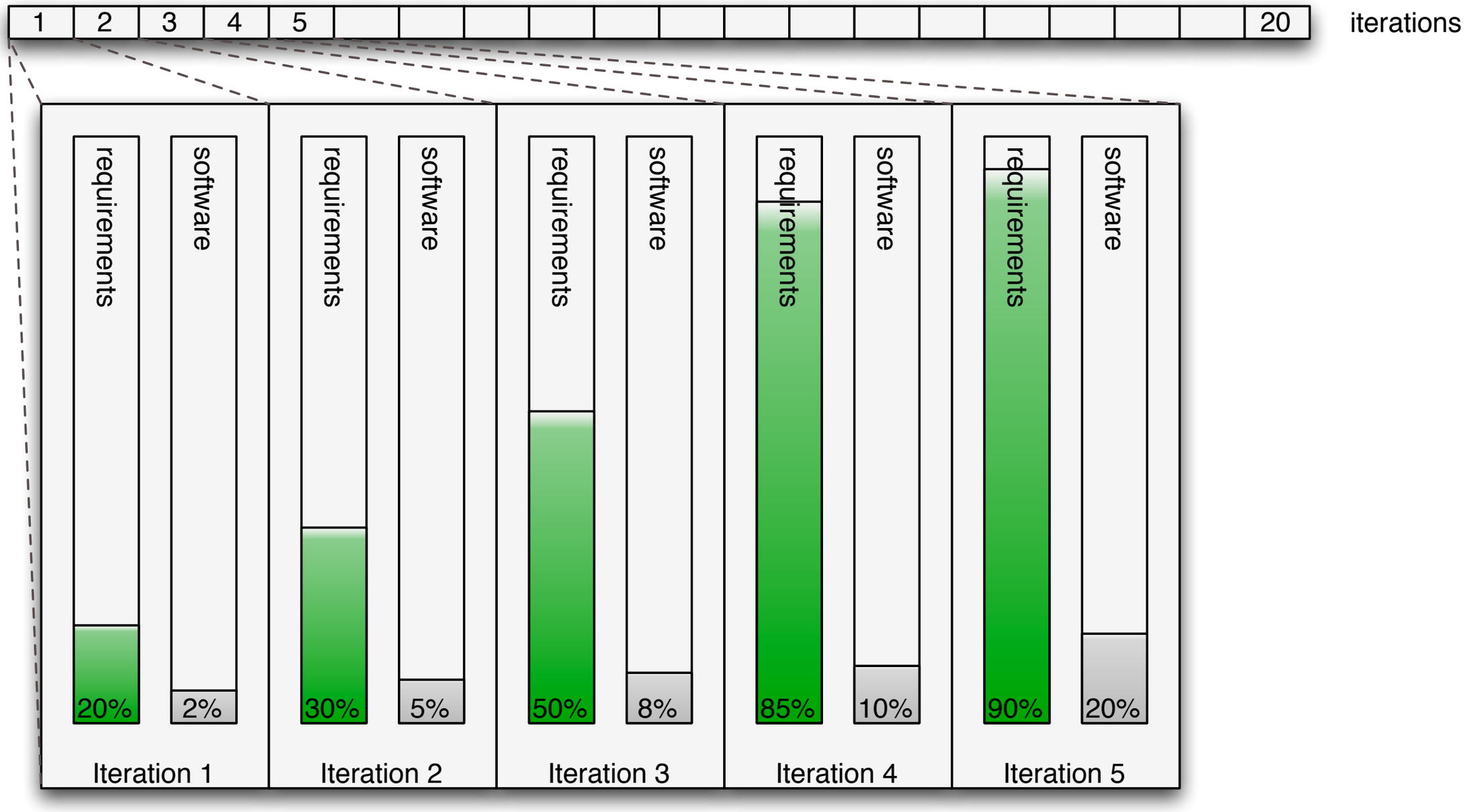
A Very First Glimpse

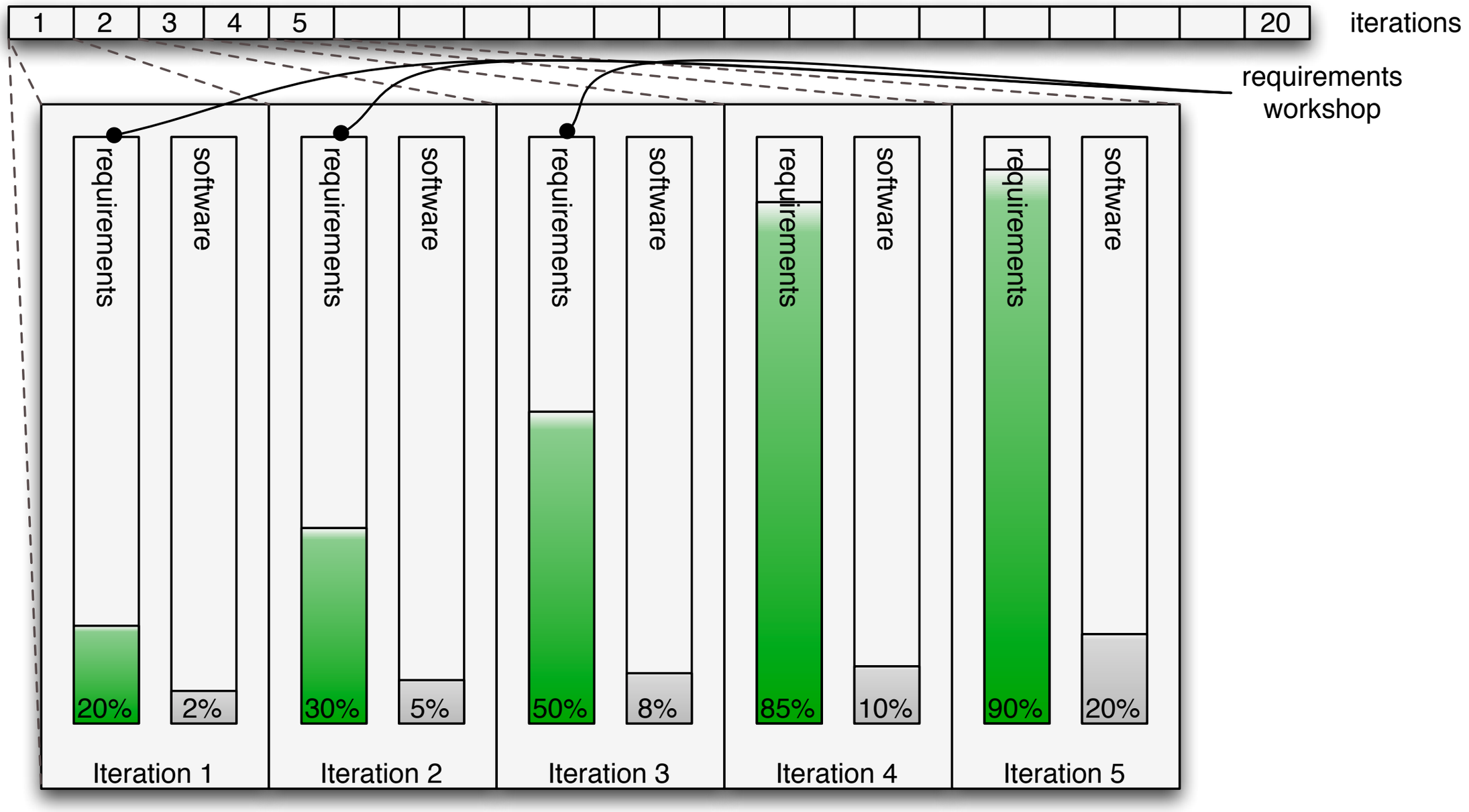
TECHNISCHE UNIVERSITÄT DARMSTADT

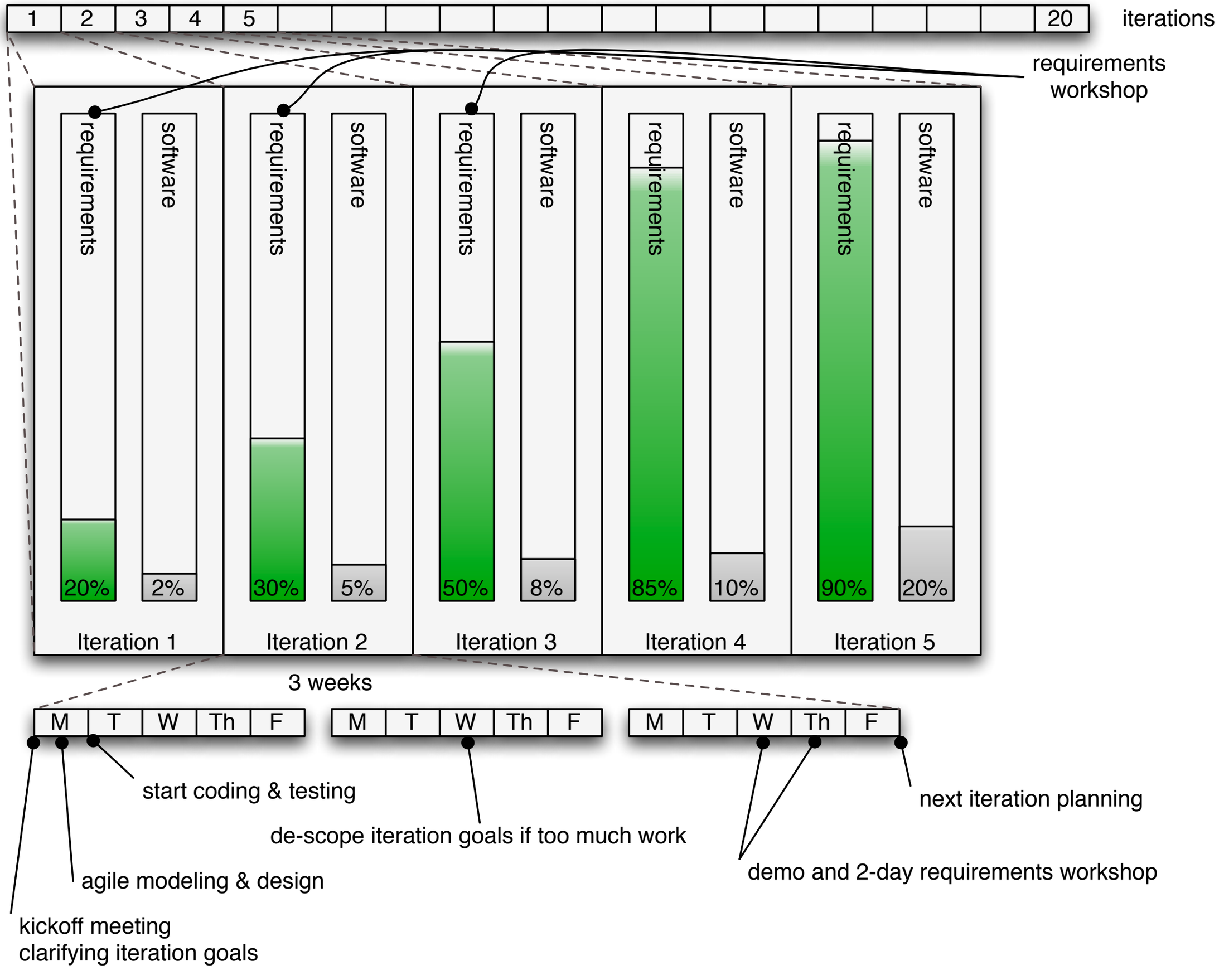
Unified Process - Phases

1. **Inception (~dt. Konzeption)**
   Feasibility phase, where just enough investigation is done to support a decision to continue or stop
2. **Elaboration (~dt. Entwurf)**
   The core architecture is iteratively implemented; high risks are mitigated
   (mitigate =dt. mildern / abschwächen)
3. **Construction (~dt. Konstruktion)**
   Iterative implementation of remaining lower risk and easier elements, and preparation for deployment
4. **Transition (~dt. Übergabe)**
   Beta tests, deployment


1
2
3
4
5
20
iterations


1
2
3
4
5
20
iterations
requirements
software
20%
2%
Iteration 1
30%
5%
Iteration 2
50%
8%
Iteration 3
85%
10%
Iteration 4
90%
20%
Iteration 5


1
2
3
4
5
20
iterations
requirements workshop
requirements
software
20%
2%
30%
5%
50%
8%
85%
10%
90%
20%
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5


1
2
3
4
5
20
iterations
requirements workshop
requirements
software
20%
2%
30%
5%
50%
8%
85%
10%
90%
20%
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
3 weeks
M
T
W
Th
F
start coding & testing
de-scope iteration goals if too much work
next iteration planning
agile modeling & design
demo and 2-day requirements workshop
kickoff meeting
clarifying iteration goals

General Practices

- Tackle high-risk and high-value issues in early iterations
- Continuously engage users for evaluation, feedback, and requirements
- *Build a cohesive core architecture in early iterations*
- Continuously verify quality; test early, often, and realistically
- Apply use cases where appropriate
- Do some visual modeling
- Carefully manage requirements
- Practice change request and configuration management

Extreme Programming

In the following we discuss (only) those principles that are now commonly used across most projects.

Extreme programming

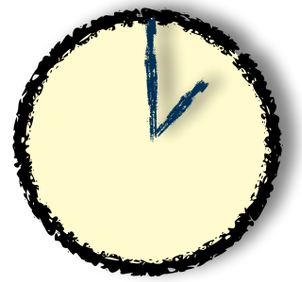
is made up of a set of simple, interdependent **practices**.

Practices = dt. Verfahren / Verfahrensregeln

Extreme Programming - Practices

User Stories

Requirements are talked over with the customer but only a few words that reminds everybody of the conversation are written on an index card along with an estimate.

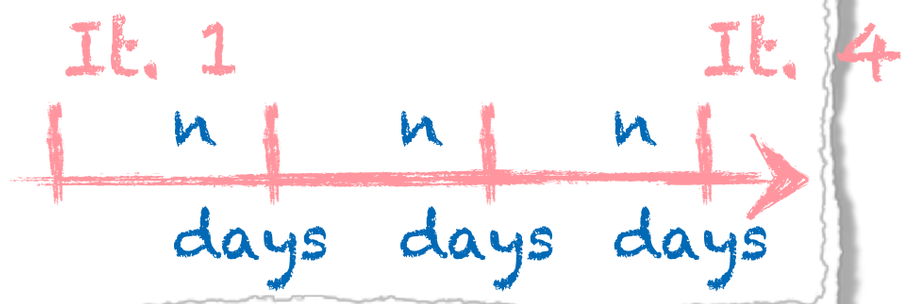


Extreme Programming - Practices

Short Cycles

Working software is delivered every, e.g., two weeks (an iteration); the delivered software may or may not be put into production.

Iterations are timeboxed - date slippage is illegal; if you cannot complete all tasks scheduled for the iteration remove some.



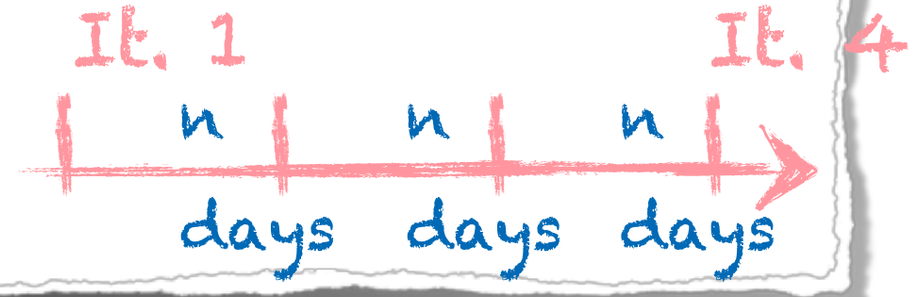
Extreme Programming - Practices

Short Cycles

Iteration Plan

During each iteration the user stories and their priorities are fixed.

The customer selects the user stories they want to have implemented. The number of stories is limited by the budget, which is set by the developers.

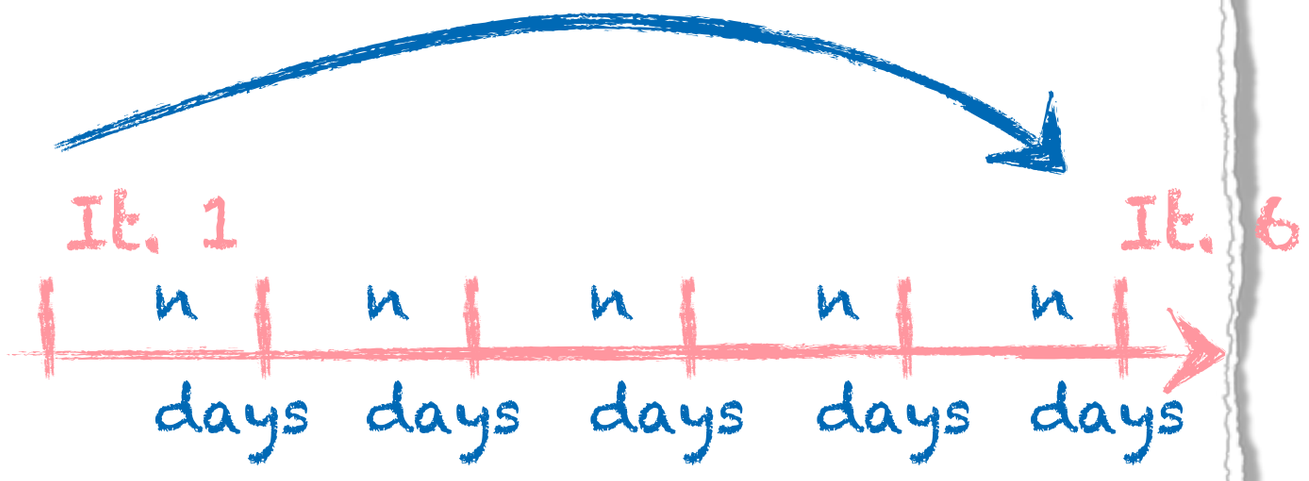


Extreme Programming - Practices

Short Cycles

Release Plan

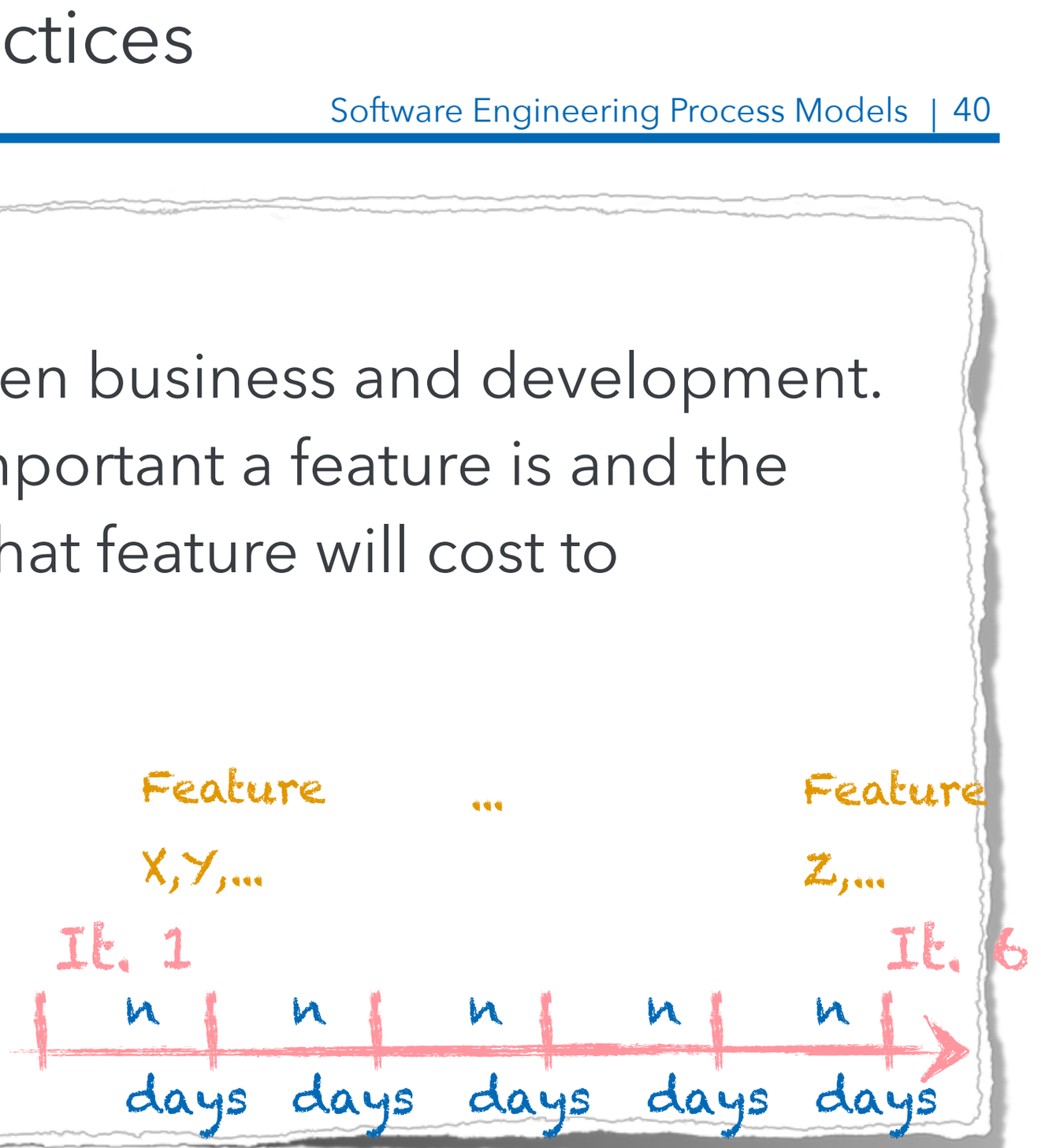
Maps out approx. six iterations. Can always be changed.



Extreme Programming - Practices

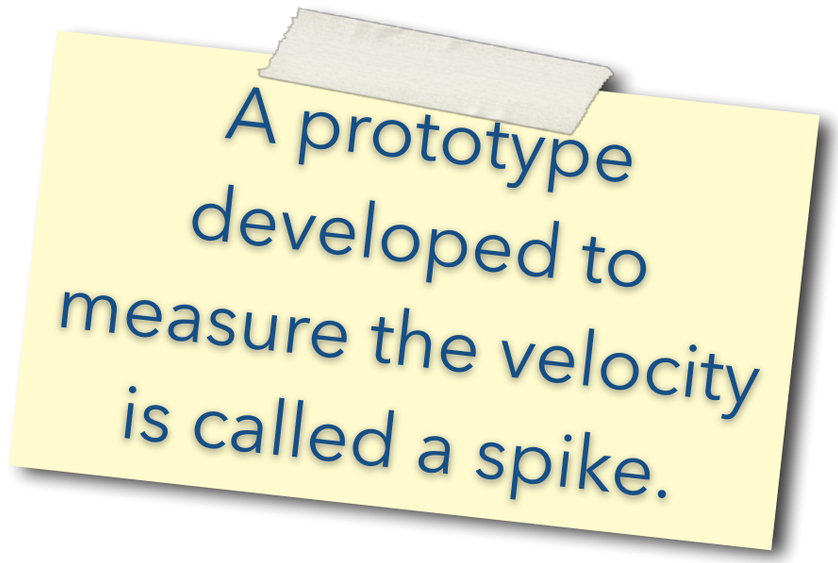
The Planning Game

Division of responsibility between business and development. Business people decide how important a feature is and the developers decide how much that feature will cost to implement.



Extreme Programming - Planning

- **Initial Exploration (Start of the Project)**
  - Developers and customers try to identify all *significant* user stories; i.e., they do not try to identify all stories
  - The developers estimate - relative to each other - the stories by assigning story points; a story with twice as much points as another story is expected to take twice as long to implement
- To know the true size we need the velocity
  (velocity = time required per story point)
  The velocity will get more accurate as the project proceeds; initially it is just guessed based on “experience”.



Extreme Programming - Planning

- **Iteration Planning**
  - The customer picks the stories for the iteration
  - The order of the stories within the iteration is a technical decision
  - The iteration ends on the specified date (timeboxed), even if all stories aren't done
  - The estimates for all the stories are totaled and the velocity for that iteration is calculated
  - The planned velocity for each iteration is the measured velocity of the previous iteration

Extreme Programming - Planning

- **Task Planning**
  - At the start of each iteration the developer and customers get together to plan
  - The stories are broken down into tasks which require between 4 and 16 hours to implement
  - Each developer signs up for tasks
    - A developer can choose an arbitrary task - even if he is not an expert

The Planning Game - “Planning Poker”

- The Product Manager provides a short overview. The team is given an opportunity to ask questions and discuss to clarify assumptions and risks.
- Each individual lays a card face down representing their estimate. Units used vary - they can be days duration, ideal days or story points.
- Everyone calls their cards simultaneously by turning them over.
- People with high estimates and low estimates need to justify their estimate and then discussion continues.
- Repeat the estimation process until a consensus is reached.

For further details go to: https://en.wikipedia.org/wiki/Planning_poker

The Planning Game - Playing “Planning Poker”

- Estimate the effort required to implement the following functionality:
  - Read an “annotated Properties” file and return a map that contains the specified and validated properties.
    - Each line is either empty, starts with a “#” if it is a comment, or uses the following pattern:
      ‘[’<TYPE>’]’<KEY> ‘=’ <VALUE>
  - If a failure occurs while parsing a line, the line is ignored and parsing continues with the next line.
  - After parsing the complete file, a map is returned with the validated properties; all lines that cannot be parsed or fail validation are also returned.

Extreme Programming - Practices

Acceptance Tests

Details of the user stories are captured in the form of acceptance tests.

Acceptance tests (which are typically black-box tests) are written before or concurrent with the implementation of a user story.

Once an acceptance test passes, it is added to the set of passing acceptance tests and is never allowed to fail again.

Extreme Programming - Practices

Pair Programming

The code is written by pairs of programmers; one types the code and the other member watches the code being typed - the keyboard is moved between the developers. The pairs change after half a day to make sure that the knowledge is spread.

An extension of the idea is “Mob Programming”

Extreme Programming - Practices

Refactoring

Do frequent refactorings to avoid that the code “rots” due to adding feature after feature.

Refactoring means improving the structure without changing behavior.

Extreme Programming - Practices

Test-Driven Development

All code is written to make failing (unit) tests pass! Having a (very) complete body of test cases facilitates refactorings and often (implicitly) leads to less coupled code.

Extreme Programming - Practices

Continuous Integration

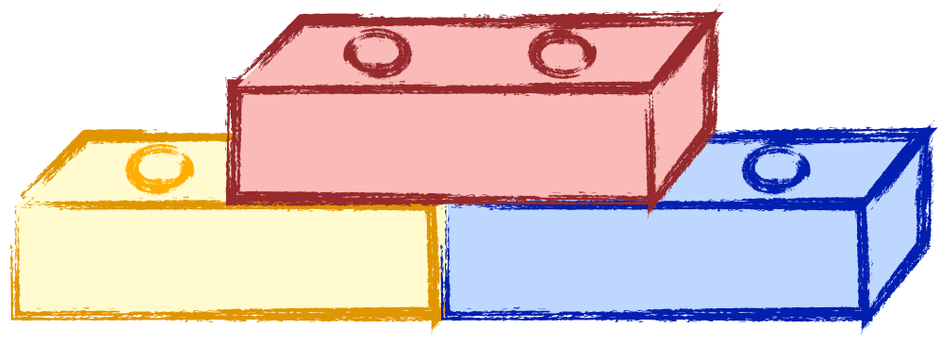
Programmers check in their code and integrate several times per day; non-blocking source control is used. After check-in the system is build and every test (including running acceptance tests) is run.

Extreme Programming - Practices

Simple Design

Make the design as simple and expressive as possible. Focus on the current set of user stories; don't worry about future user stories.

E.g. only add the infrastructure when a story forces it.

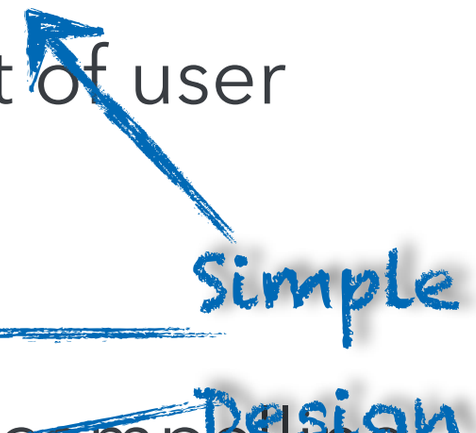


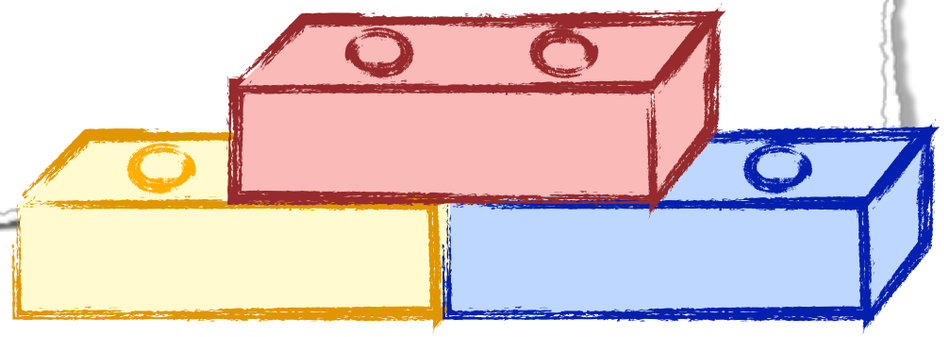
Extreme Programming - Practices

Consider the simplest thing that could possibly work!
Find the simplest design option for the current set of user stories.

You aren't going to need it!
Add infrastructure only if there is proof or at least compelling evidence.

Once and only once; don't tolerate code duplication; eliminate code redundancies by creating abstractions. Employ patterns to remove redundancies.





```
boolean isInitValueValid(long v) {
  if ((v < Integer.MIN_VALUE) || (v > Integer.MAX_VALUE)) {
    return false;
  }
  return true;
}

boolean isInitValueValid(int v) {
  if ((v < Integer.MIN_VALUE) || (v > Integer.MAX_VALUE)) {
    return false;
  }
  return true;
}
```

OpenJDK8

com.sun.jmx.snmp.SnmpInt; lines 191-212

Example: User Stories for a Web Application

James Newkirk and Robert C. Martin

Extreme Programming in Practice; Addison Wesley, 2001

Example: User Stories for a Web Application

Estimates (upper right corner) are given in ideal days in this case

one day

Some pages trigger the login mechanism and some don't.

The list of pages that do/don't is dynamic.

And the mechanism is triggered once per session.

Example: User Stories for a Web Application

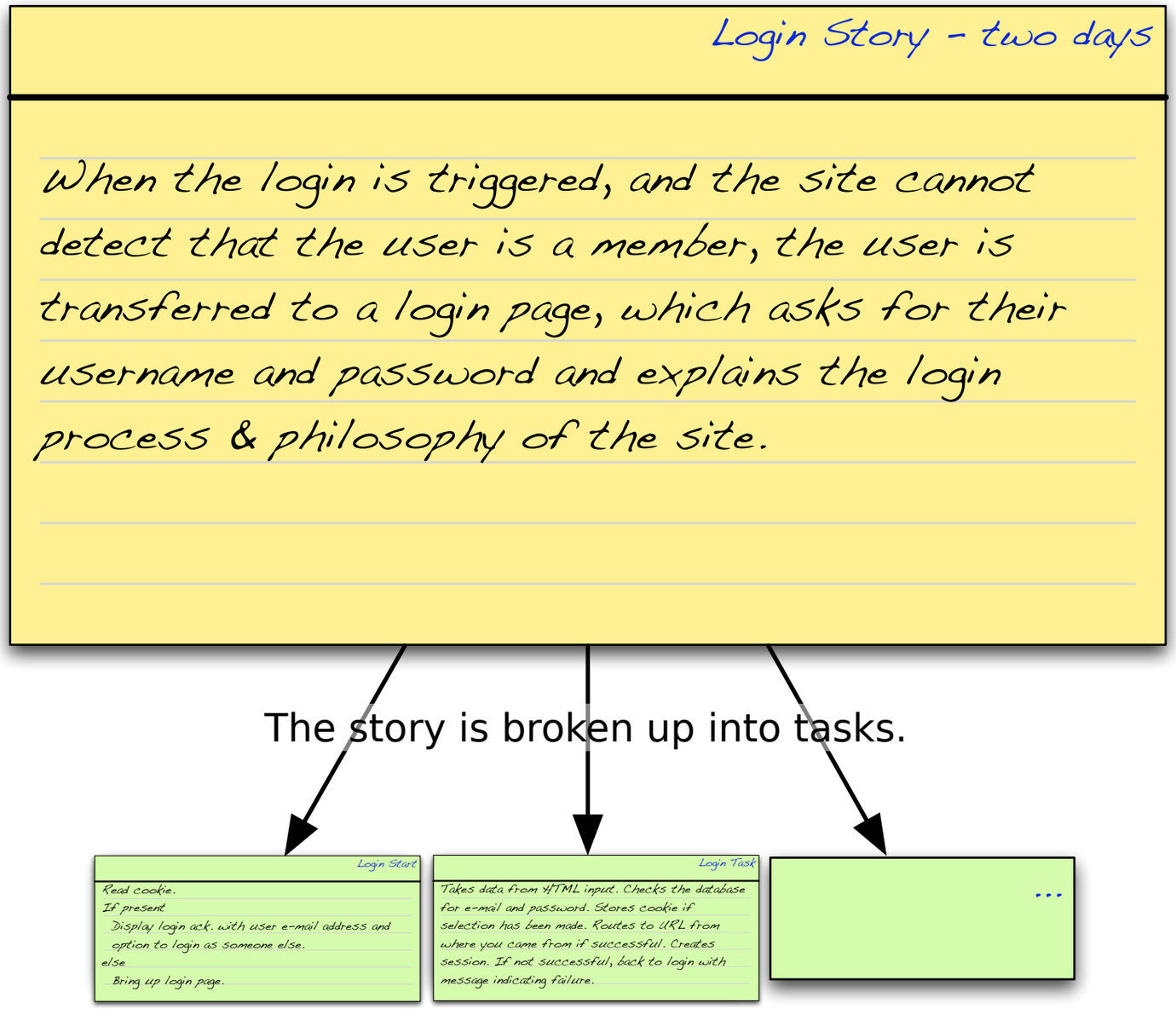
Non-implementable user stories

Constraint

The system will not pop up a window that could be interpreted as a pop-up ad.

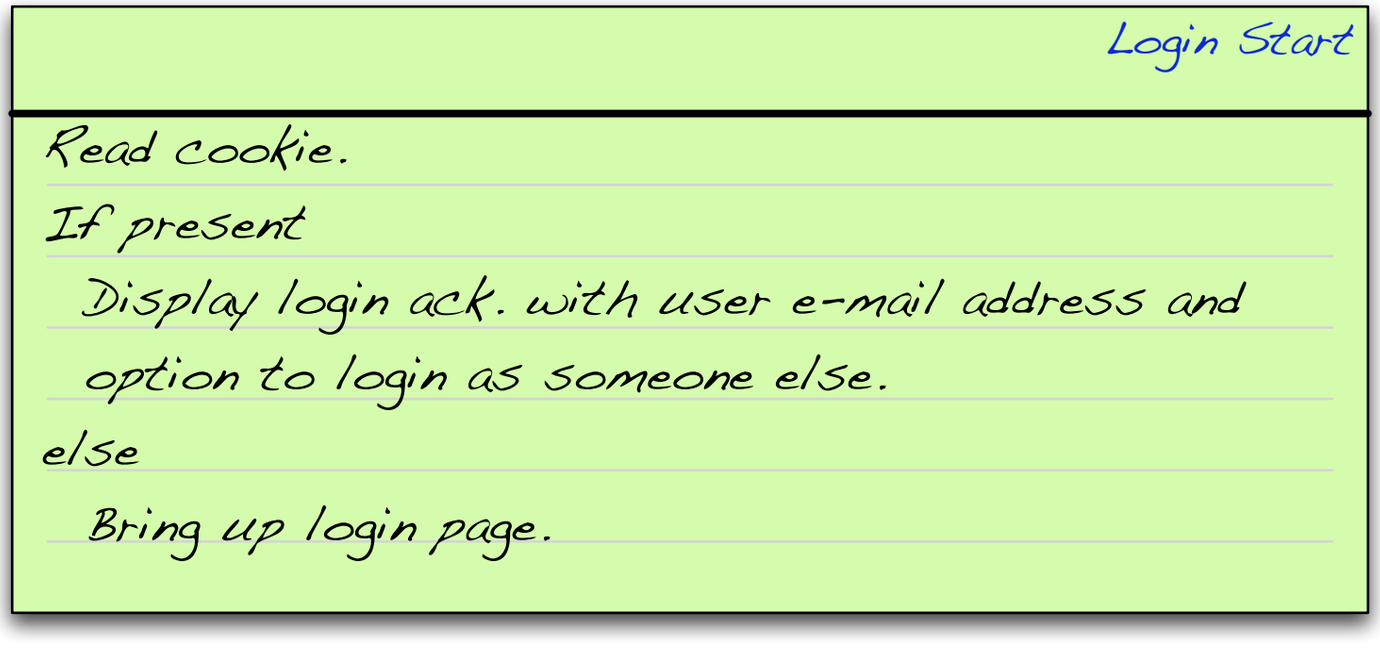
Example: User Stories for a Web Application

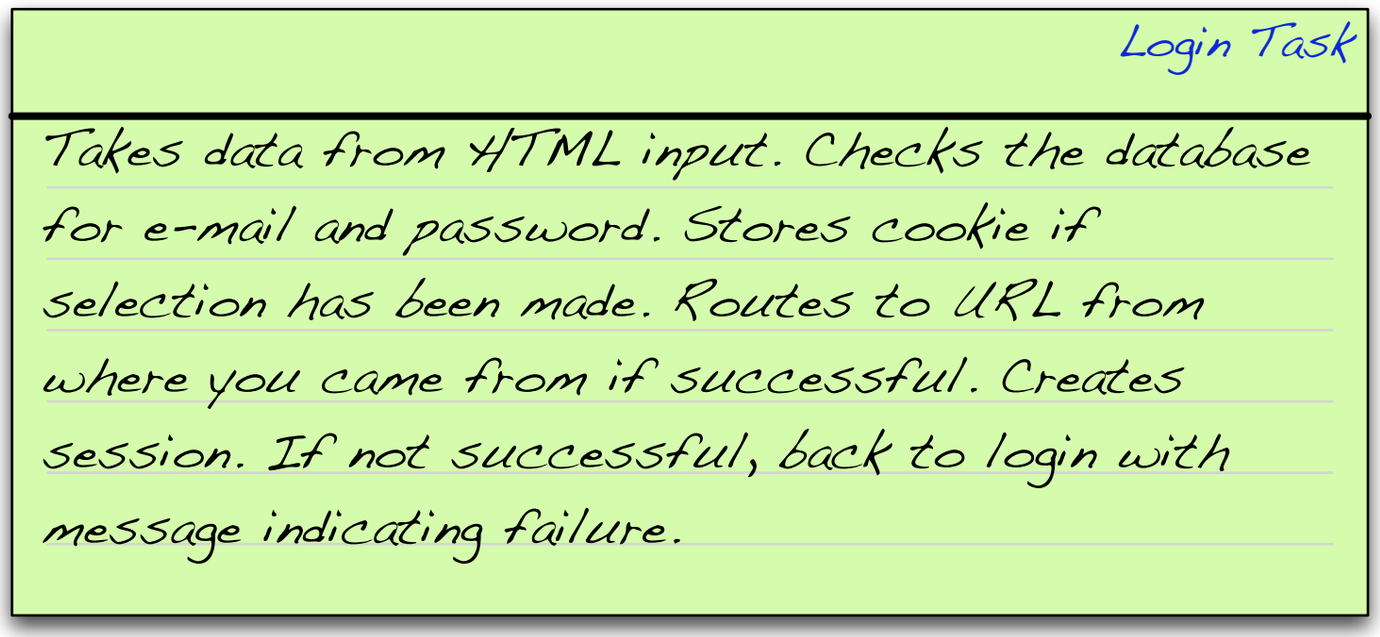
Breaking down stories into tasks.



Example: User Stories for a Web Application

Breaking down stories into tasks.





Principles of Good Stories

- Stories must be understandable to the customer
- Each story must provide something of value to the customer
- Stories need to be of a size that you can build a few of them in each iteration
- Stories should be independent
- Each story must be testable

INVEST

Independent, **N**egotiable, **V**aluable, **E**stimable, **S**ized appropriately, **T**estable

Established Templates for Writing User Stories

- Long template:
  "As a <role>, I want <goal/desire> so that <benefit>"

- Shorter template:
  "As a <role>, I want <goal/desire>"

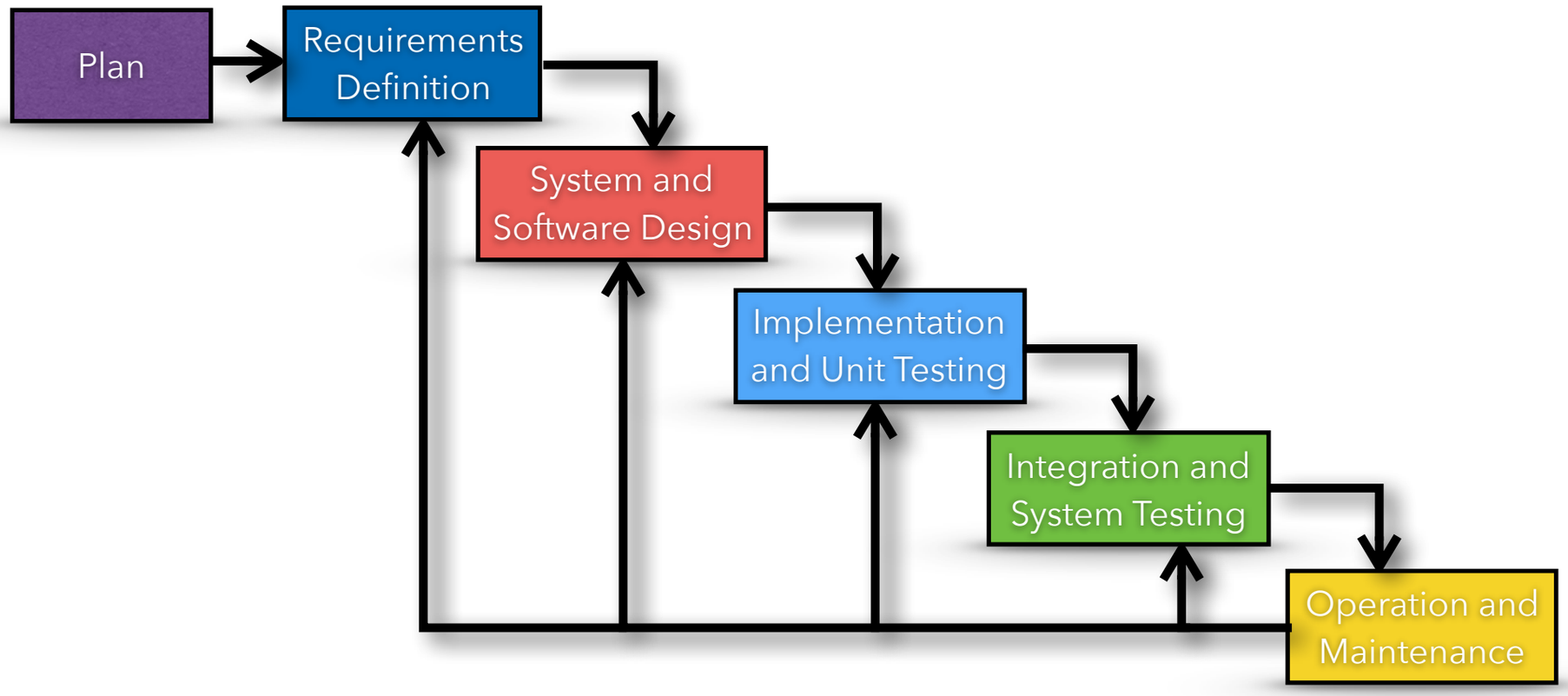
Recording User Stories - An Example

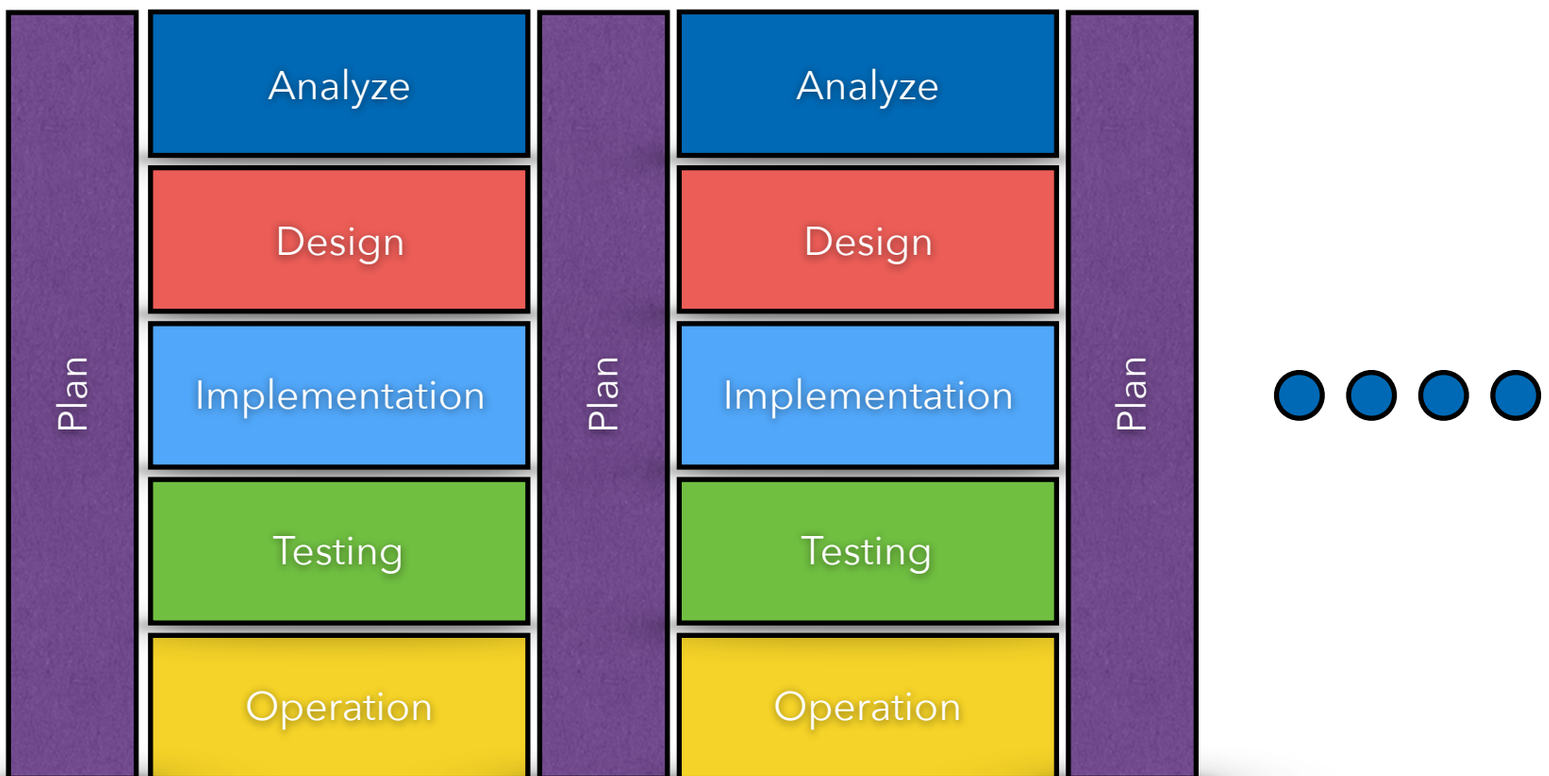
ID	2
Name	Login
Beschreibung	Als Administrator muss ich mich am System mittels Benutzername und Passwort authentifizieren können, um Änderungen vornehmen zu
Akzeptanzkriterium	Der Dialog zum Einloggen wird korrekt angezeigt und es ist möglich sich als Administrator zu authentifizieren. Ungültige Eingaben werden ignoriert und normale Nutzer erhalten nicht die Rolle “Administrator”.
Geschätzter Aufwand (Story Points)	3
Entwickler	Max Mustermann
Umgesetzt in Iteration	2
Tatsächlicher Aufwand (Std.)	12
Velocity (Std./Story Point)	4
Bemerkungen	/

Scrum a Brief Overview

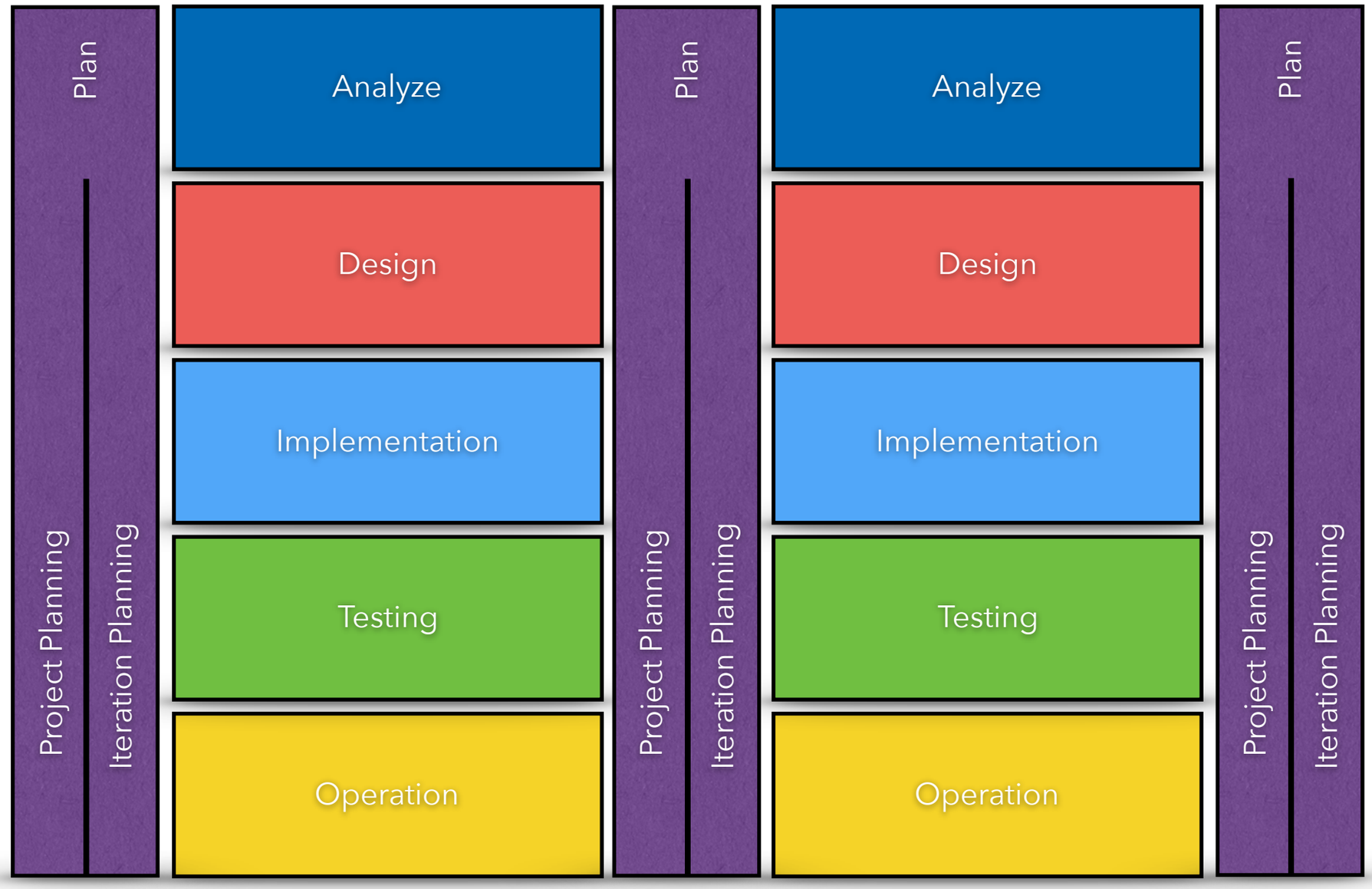
- Scrum is a project management framework
- Scrum employs an iterative, incremental approach to optimize predictability and control risk

Scrum vs. Waterfall

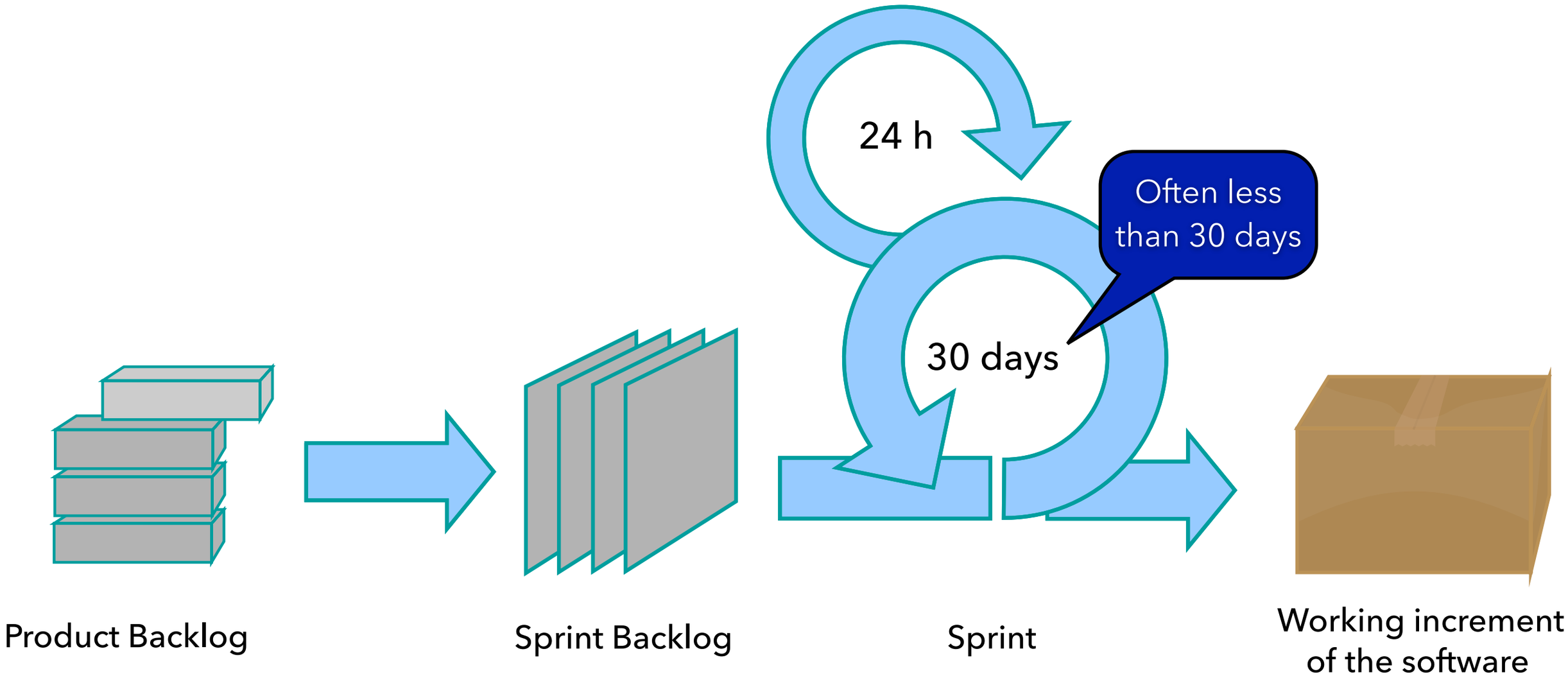




Scrum Planning



Scrum a Brief Overview



"Scrum process" by Lakeworks - Own work. Licensed under GFDL via Commons - https://commons.wikimedia.org/wiki/

Scrum a Brief Overview - Roles & Responsibilities

- **Product Owner**
  - *m*anages the Product Backlog
- **Development Team**
  - delivers working software (i.e., software that is done - w.r.t. a reasonable definition of “done”)
  - manages itself
- **Scrum Master**
  - takes care of the process and
  - ensures that the process is followed

Scrum a Brief Overview - Events I

- **Sprint**
  - time boxed iteration
  - all development is done within a sprint
  - starts with a sprint planning and ends with the sprint review
- **Sprint Planning**
  - identify the tasks/features from the product backlog to work on during the sprint (determine the sprint backlog); done in collaboration between the product owner and the development team
  - do design work

Scrum a Brief Overview - Events II

- **Daily Scrum**
  - a short, time boxed meeting to do just-in-time planning
- **Sprint Review**
  - the development team and the product owner review the results
  - the goal is to update the product backlog based on the results
- **Retroperspective**
  - inspects how the last sprint went to improve it (and actually acts accordingly)

Scrum a Brief Overview - Artifacts

- **Product Backlog**
  - the prioritized list of things that need to be done (maintained by the product owner)
  - transparent
  - the priority is determined by considering the business value and the risks involved
  - high-priority things should be decomposed such that the things (e.g., development tasks) are actionable
- **Sprint Backlog**
  - actionable items from the product backlog

Different types of systems need different development processes.

E.g. software used in an aircraft has to be developed using a different development process as an e-commerce web page. An operating system has to be developed differently from a word processor.

In large software systems different parts may be developed using different process models.

The one software process does not exist.

Processes have to exploit the capabilities of the people in an organization and the specific characteristics of the systems that are being developed.

Summary

The goal of this lecture is to enable you to systematically carry out small(er) software projects that produce quality software.

- To systematically develop software, you have to follow a well-defined process that suites the needs of the project under development.
- It is practically impossible to work out all requirements right at the beginning of a project.

- The goal of this lecture is to enable you to systematically carry out small(er) commercial or open-source projects.