LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Agile Software Development with Scrum

An Iterative, Empirical and Incremental Framework for Completing Complex Projects

Dr. Andreas Schroeder

(based on slides of Dr. Philip Mayer and Annabelle Klarl)

CHAOS Report 2009

Completion of projects:

- 32% success
- 44% challenged
- 24% impaired

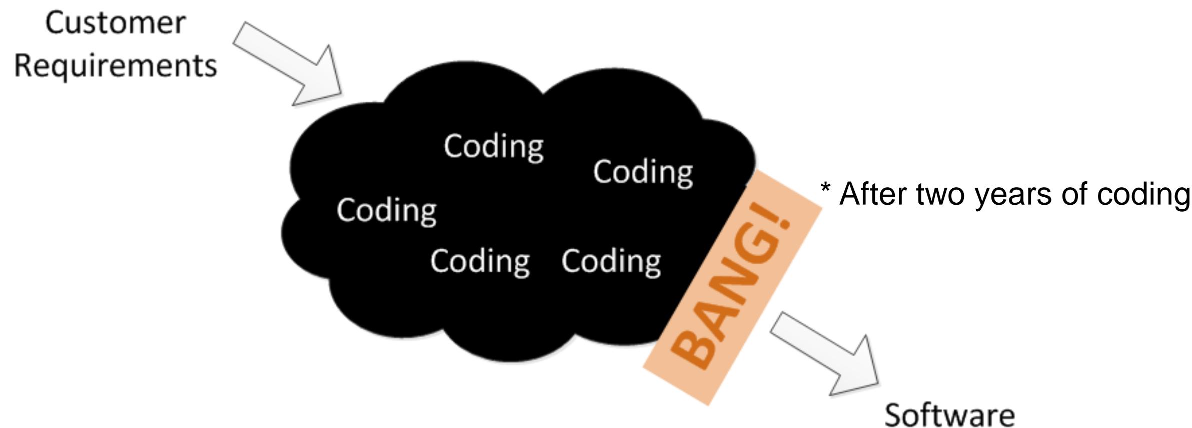
2/3 of all projects don’t satisfy their expectations

Fail factors (excerpt):

- Incomplete requirements
- Changing requirements
- Little involvement of the customer
- Low support by the management

Sequential paradigm

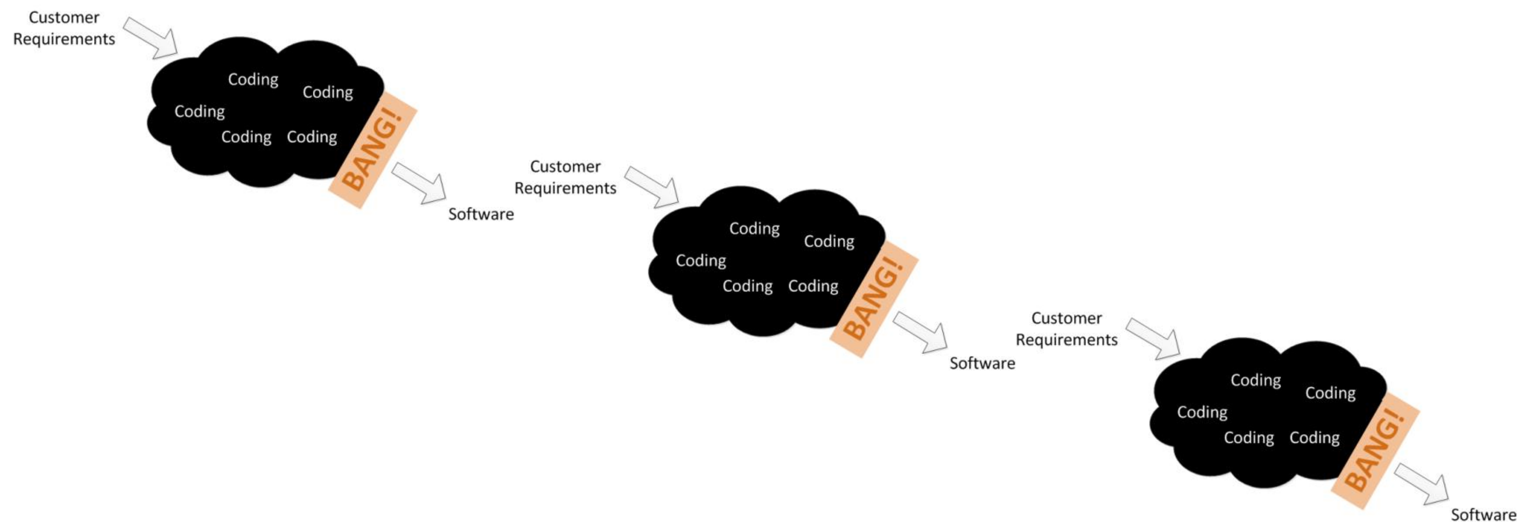
- The **Big Bang** approach to software does **NOT** work:



- No interaction with the customer in the black cloud!
- **The problem**:
  - Requirements might have been misunderstood or changed.
  - The resulting system is not what the customer wanted.

Iterative paradigm

- The **iterative** approach to software does **not** work **either**:



- Requirements are captured while product is unknown.
- Requirements Phase is exaggerated until no time for implementing is left.

Change is the only constant in SW development

- "Expect the unexpected!"
  Agile methods build on the ability to react to change.
- "Get it working!"
  Agile methods deliver working software frequently.
- "Please the customer!"
  Agile methods build on openness and communication.

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.

Why agile software development?

Software development is like new product development, not like manufacturing

- Manufacturing: building the same model again and again
- Software development: creating something new

We need:

- Research and Learning
- Creativity
- Subtle Control and Self-organization

Examples for agile methods

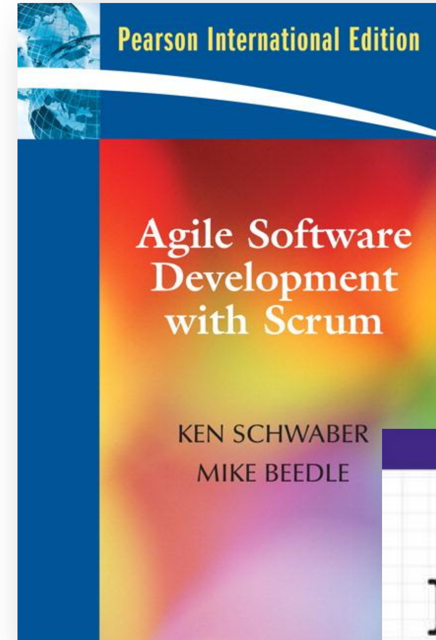
A multitude of agile processes has been introduced

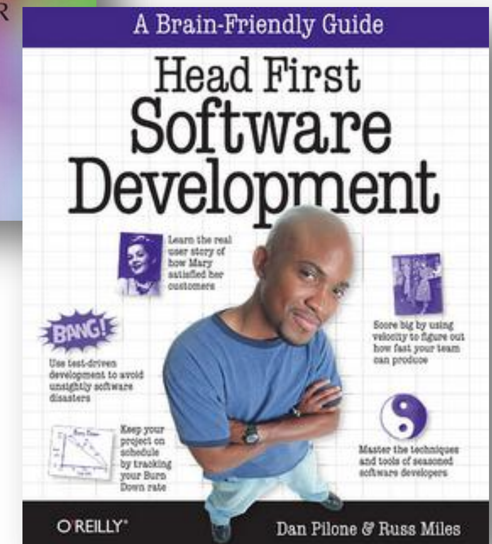
- Kanban www.kanbanblog.com/explained
  - Help work to flow (= continuously deliver)
  - Make work visible (= show what is going on)
  - Limit work in progress (= promote quality, focus and finishing)
- XP (eXtreme programming) www.extremeprogramming.org
- Scrum www.scrum.org
- …

Agile methods in SWEP

- Scrum (mainly)
- XP
- Head First Software Development Process

- The Scrum process
  - follows the agile manifesto
  - is intended for groups of 7
  - consists of simple rules and is thus easy to learn





Agenda Part I

- **Scrum Overview**
  - The source of Scrum
  - The three legs
  - The big picture
- **Scrum Roles**
  - About pigs and chickens
  - Your friend in need: the Scrum Master
  - To whom everybody listens: the Product Owner
  - „We sink and swim together“: the Scrum Team
- **Capturing and Managing Requirements**
  - Understanding the Customer: Release Planning Meeting
  - Structuring Requirements in Product Backlog Items
  - Priority and Estimation

Agenda Part II

- **Planning and Controlling the Process**
  - Deciding on Items for a Sprint
  - Daily Scrum Meeting: know where you stand
  - Pleasing the customer with a demo
  - Learning from the process
- **Development in detail**
  - Testing
  - Managing Bugs
  - Development in a Team
  - Software Design
- **Scaling Scrum**
- **Conclusion**

Part I/VI. Scrum Overview

Scrum in rugby

Strategy for getting the ball back into play

www.andrewgoss.net/sport.html

Scrum as an agile method

„a holistic or „rugby“ approach – where a team tries to go the distance as a unit, passing the ball back and forth“

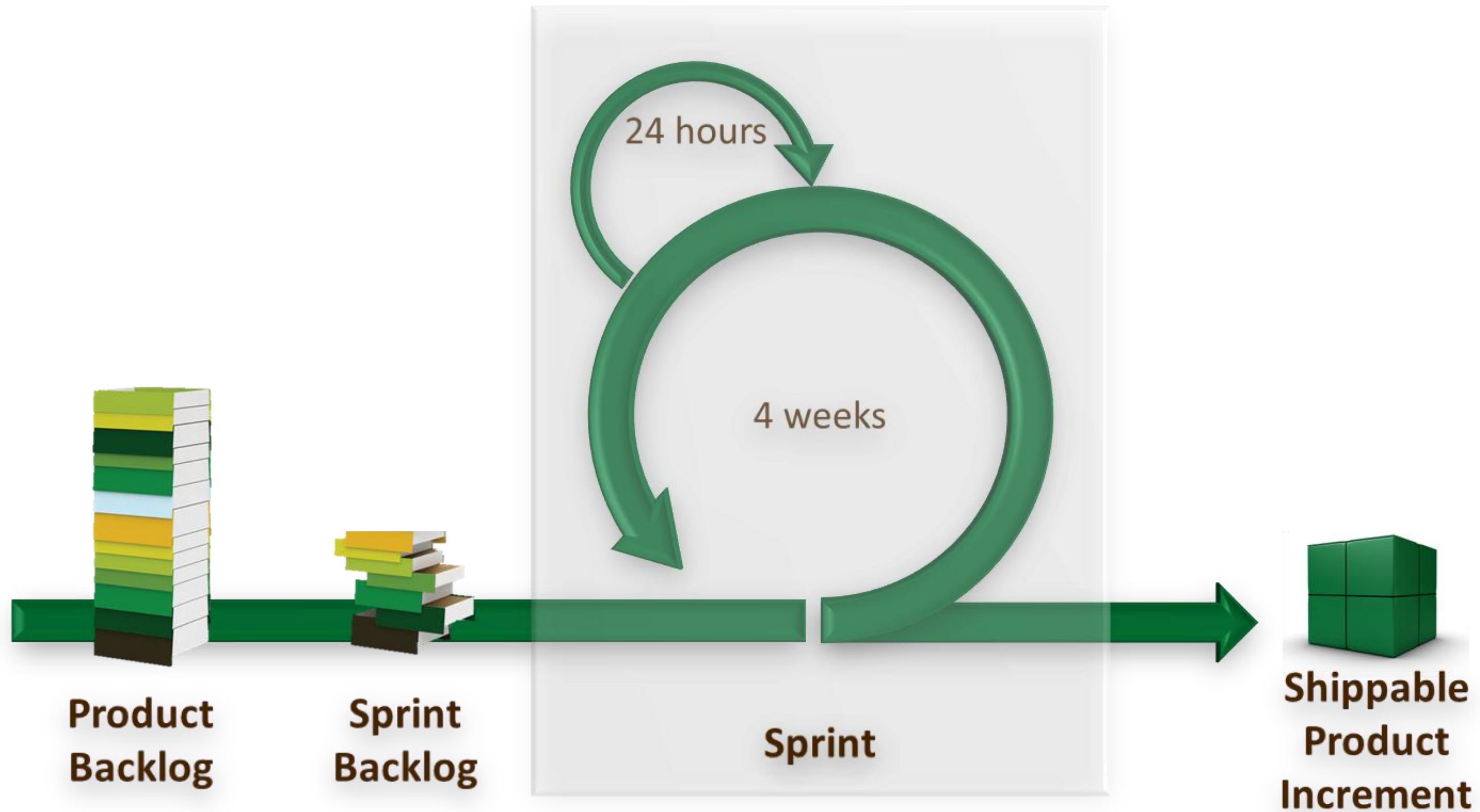
Takeuchi, H. & Nonaka, I. The new new product development game. *Harvard Business Review* **64**, 137-146 (1986).

Part I – The three legs of Scrum

Scrum is grounded in **empirical process control theory** and is therefore not guided by a fixed project plan, but by

- **Transparency**
  „Everything can be seen by everybody."
- **Inspection**
  „The process is continuously monitored."
- **Adaption**
  „Feedback mechanisms are the heart of Scrum."

Part I – The big picture



An iterative, empirical and incremental framework

Part I – Product Backlog and Sprints

- **Product Backlog**: **Everything** with respect to the product or the process, that **anyone** is interested in, is represented in the Product Backlog.
- **Sprints**: A Sprint is a short timeframe of about four weeks for working on the Sprint Backlog - a **fixed** subset of the Product Backlog, producing a **working piece of software**.
- **Daily Scrum Meetings**: The current **progress** of work and any **impediments** are revealed in this daily time-boxed meetings.

Part I – Summary

Software development is about shipping software that brings the customer's ideas to life.

Which means:

- **Shipping software**: The software must be completed, executable and delivered – **on time** and **on budget**.
- **Customer's Ideas**: The customer has a vision of his product. The developer must be flexible enough to extract that image, **implement it** and nevertheless **react to changes**.

Part II/VI. Scrum Roles

Part II – About pigs and chickens

A chicken and a pig are together when the chicken says,

„Let‘s start a restaurant!“

The pig thinks it over and says,

„What would we call this restaurant?“

The chicken says, **„Ham n‘ Eggs!“**

The pig says,

„No, thanks. I‘d be committed, but you‘d only be involved!“

- Pigs: Everyone with total commitment to the project
- Chickens: Everyone else who is interested in the project

Part II – Scrum Master

The Scrum Master is **responsible for the success** of Scrum.

- Responsibilities:
  - Institutes a Product Owner
  - Forms a Scrum Team
  - Assists all Planning Meetings
  - Ensures that **Scrum values, practices and rules are enforced**
  - **Removes any impediments**
- Although being a management role, the Scrum Master should be **your friend in need**.

Part II – Product Owner

The Product Owner is officially **responsible for the product**.

- Responsibilities:
  - Represents the customer
  - Maintains the **Product Backlog**
    - Registers new Items
    - Prioritizes Items
    - Get the estimates for Items
  - Makes the Product Backlog visible to everyone
- **Developers only listen to the Product Owner.**

Part II – Scrum Team (I)

The Scrum Team **commits to achieving a Sprint Goal**.

- 7 (+-2) developers
  - < 5 developers impose skill constraints
  - > 9 developers induce complex coordination
- Responsibilities:
  - Decides on a Sprint Goal in compliance with the Scrum Master and the Product Owner
  - Commits to turn the selected set of the Product Backlog into a working product during a Sprint
  - Has **full authority how** to achieve the Sprint Goal

Part II – Scrum Team (II)

"We swim and sink together"

- The whole team is responsible for the whole product
- Normally full time members
- No titles
- All-round developers
  – or at least willing to assist each other

Team composition may only change at the **end** of a Sprint, but experts can be invited to assist the development!

Part II – Chickens

- Everyone else who is interested in the project
  - **Customer**
  - Management
  - Other Scrum Teams (working on the same project, depending projects or totally different projects)
  - Observers

- Chickens are **not allowed to influence** the work of the Scrum Team **during a Sprint**!

Part II – Summary

- Pigs are **committed** to the project.
  - The Scrum Master enforces the Scrum rules.
  - The Product Owner manages the Product Backlog.
  - The Scrum Team is committed to the Sprint Goal.

- Chickens are only **involved** into the project.
  - Scrum Teams must not listen to chickens.
  - Chickens are only allowed to consult.

Part III/VI. Capturing and Managing Requirements

Part III – Release Planning Meeting

- The **Product Vision** is the customer’s mental image about his software.
- The goal of the Release Planning Meeting is „How can we turn this vision into a winning product?“
  - Overall features and functionalities
  - Major risks
  - Probable delivery date and cost

- But how do we extract the correct requirements from the Product Vision?

Part III – Product Backlog Items

- In Scrum, requirements are captured in the form of **Product Backlog Items** (PBI).
- For the SWEP, we use **User Stories** and **Issues** as PBIs.
  - A User Story captures **one thing** (and one thing only) that the software needs to do for the customer.
    - A User Story has a **title** and a **short description**
    - The description should fit on a DIN A6 index card (if it is too long, it needs to be split in two)
  - An Issue captures **one thing** that is hard to mold into a User Story e.g. software quality issues like bugs and safety, security or performance as well as documentation matters.

Part III – Capturing User Stories

- User Stories are **customer-oriented**.
  - User Stories are written with and for the customer
  - They must be written in a language the customer can understand

- Techniques for capturing requirements
  - Blueskying: brainstorming with the customer
  - Role playing: developer acts as the new software
  - Observation: developer watches the customer do the tasks to be supported by the new software

Part III – Examples

- Good Story (customer-level):

View Games

Users should be able to see games which are open for participation.

- Bad Story (too technical):

Use MySQL as database

The database will be based on mySQL as it is a stable and open-source solution.

- All User Stories and Issues make up the Product Backlog.

- The Product Backlog is **never** complete!
  - **Everyone** (pigs and chickens) may add items.
  - The Product Backlog evolves during the project by adding or changing requirements.

- The Product Owner additionally assigns a **priority** to each Product Backlog Item.
- Furthermore, he gets an **estimate** from the Scrum Team how long it will take to implement it.

Part III – Prioritizing

The Product owner **prioritizes** the Product Backlog Items in compliance **with the customer**.

- Important ones get a higher priority and must be implemented first.
  - Priorities should be taken out of the set of **{10,20,30,40,50}** with 10 being most important .
  - Priorities are added to the Product Backlog Items.
- Priorities for Product Backlog Items **might change** depending on estimates or changing requirements.

Part III – Estimating (I)

- The Scrum Team **estimates** PBIs **without the customer**.
- Estimation means guessing the number of hours for constructing each PBI.
  - The User Story or Issue is **split into tasks**.
  - A task specifies a piece of development to be carried out by **one** developer.
    - Like a User Story, it has a **title** and a **description.**
    - **Usually attached to a User Story.**
  - Estimate: **How long** will it take it get the task done?
  - The combined estimates are the overall estimate for the PBI.

View Games

Users should be able to see games which
are open for participation.

Create table for games in the DB

Create a UI for browsing games

Allow retrieval of games from server

Part III – Estimating (II)

- The whole Scrum Team is responsible for the project.
- Everybody should, in principle, be able to implement each functionality. Thus, estimation takes **everybody** into account!
- Each estimate should include time for
  - Design
  - Code and Document
  - Test and Review
  - Integration and Delivery
- To arrive at a number **everybody is comfortable with** we use Planning Poker.

Part III – Planning Poker (I)

- Planning Poker
  - A certain task is presented.
  - Every developer thinks about the task and how long it will take **himself** to implement it, all things considered.
  - Every developer **privately** chooses a card from the deck with cards for 0, ½, 1, 2, 3, 5, 8, 13, 20, 40 and 100 hours.
  - All cards are **simultaneously** uncovered.
  - High and low estimates are discussed.
  - The estimation process is repeated until convergence.
- **Note:** A task should take about 6 hours to implement, so that a User Story takes a couple of days.

Part III – Planning Poker (II)

- The goal is convergence.
  - The team **must come up with a single estimate.**
  - If the estimates differ a lot, this indicates (probably) hidden assumptions and less confidence.
- Thus, a second goal is to uncover **assumptions**
  - ...about what is part of a story and what is not
  - ...about the skills required or the need to acquire them first
  - ...about the complexity of the task
- This might require asking the customer for clarification.
- And, a third goal is to **transfer knowledge.**

Part III – Estimating (III)

- Meaningful estimation requires **knowledge** about
  - … the existing codebase
  - … the effort involved in using the libraries and technologies
- It is borderline impossible to come up with meaningful estimates if these factors are completely unknown.
- Therefore, get familiar with the technologies **before 29.04.**

Part III – Summary

- Requirements are captured as Product Backlog Items.
  - **User Stories** are customer-oriented.
  - **Issues** capture more technical things.
  - The Product Backlog is never complete.
- The Product Owner assigns **priorities** to the Items, indicating which functionality should be implemented first.
- Product Backlog Items are split into **tasks**.
- Tasks, and thereby Product Backlog Items, are **estimated.**
  - The aim is confidence by all developers
  - ...and getting rid of assumptions.

Part IV/VI. Planning and Controlling the Process

Change is the only constant in SW development

- Requirements, Estimates, and Priorities might change – but this is considered in the process and dealt with **in a controlled way**.

- **Releases**. Our process is based on releases which take about three months.
  - A release of the software is a self-contained set of functions.
- **Sprints**. Each release is split into Sprints which take about four weeks.

Part IV – Sprint

- **Fixed** period of time: four weeks
- **Fixed** set of functionality to accomplish
  - Sprint Goal
  - Sprint Backlog
- During Sprint
  - **No interferences** with the development work
  - No additional functionality
  - No new technologies
  - **Free timing** for the Scrum Team

Part IV – Sprint activities

- Sprint Planning Meeting
  - Assigning Product Backlog Items
  - Determining Velocity
- Development work
  - Holding Daily Scrum Meetings
  - Updating Whiteboard and Burn-Down-Chart
- Sprint Review
  - Demoing the piece of running software
- Sprint Retrospective
  - Learning from the past

Part IV – Sprint Planning Meeting

- Fixed timeframe: eight hours (for a four week Sprint)
- Everybody may attend.
- The goal is to decide
  - … **what** will be done
  - … **how** it will be done

- That basically means assigning Product Backlog Items to the Sprint as Sprint Backlog Items and planning how to realize them.

The Sprint Planning Meeting is the main meeting for planning the Sprint!

- Which means...
  - ... estimate and pick User Stories,
  - ... discuss realization approach (with UML sketches),
  - ... split User Stories into reasonable tasks,
  - ... assign tasks to team members.
- But: **How many Product Backlog Items fit into a Sprint?**

Part IV – Assigning Items: Reality

- In principle, the available days are four weeks i.e. 20 working days multiplied by the number of developers (e.g. 3):

$$3 \times 20 = 60 \text{ days}$$

- **However**: Estimates are based on **ideal** days or hours. Unfortunately, the real world keeps intruding with
  - Installing Software
  - Team Communication
  - Paperwork
  - Hardware breakdowns
  - Sickness and Holidays

Part IV – Velocity (I)

- **Solution**: The amount of available days is reduced by a factor, the team velocity:

$$3 \times 20 \times 0.7 = 42 \text{ days}$$

- That means, we can select Product Backlog Items with a total estimate of 42 days for the Sprint – and not more!

- As an initial factor, a value of 0.7 is assumed.
- But, the velocity is unique for each team and must therefore be monitored and changed over time:

$$\text{velocity} = \text{estimated days} / \text{required days}$$

Part IV – Velocity (II)

- How we will determine velocity
  - Track Overhead time in the Redmine tracker
  - Compute velocity based on available data

$$V = Worked / (Worked + Overhead)$$

- Reasons
  - Lab is no full-time job
  - Flexible time management
  - Empirical approach to velocity computation in our context

- During a Sprint, the Scrum Team works on Sprint Backlog Items until they are „done“.
  - Each team has its own **Definition of Done**.
  - Ours implies full functionality, no known errors/bugs, clean code, integration, tests, documentation.

- It is important to stay on track: If a User Story or task takes longer or shorter than expected, or if additional problems come up, the team must **know** about this.
- This information is gathered in Daily Scrum Meetings.

Part IV – Daily Scrum Meetings

- **Daily** at a fixed time and place: **15 minutes**
- Everybody may attend, but **only the pigs** (Scrum Team, Scrum Master and Product Owner) are allowed to speak.
- The goal is to see
  - … what was done since the last meeting
  - … what will be done before the next meeting
  - … what obstacles are in the way

- That basically means that **every** team member has to **shortly** report on these three questions.

Part IV – No Overhead!

- Some principles ensure that these meetings are productive and informative for everybody.
  - Start **sharply** at the designated time. (regardless of who is present)
  - Report **shortly** only relevant things.
  - Report on the “**what**”, not on the “how”.
  - Detailed discussion may continue **afterwards**.
  - **Stand-up** during the meeting.

- The intention is to keep the finger on the pulse of the project.

Part IV – Social responsibility

- All team members must **actively** attend.
- This enforces the **social responsibility** for everybody:
  - Honest report what has been done
  - Face-to-face promise what is done next
  - Pressure on the management to solve problems

- Scrum builds on openness and honesty!
  - **Full transparency** of all fails and delays, but also of any progress and completion
  - Only like that the Scrum Team is able to react to changes.

Part IV – Visualizing the progress

- The Whiteboard keeps track of the **current progress**.
- It shows
  - … which Sprint Backlog Items must be implemented during the Sprint
  - … which tasks are in progress
  - … which tasks have been completed during the Sprint
  - … how fast the development progress is compared to the plans for this Sprint

- Therefore, it must **always** be updated during the Daily Scrum Meetings.

Part IV – The Whiteboard

User Stories

View Games

Priority: 20
Estimation: 1 days

Server/Client Comm. Est: 2h

(User stories to be implemented in this iteration)

Tasks In Progress

Create Game Table
Est: 2h
Developer: PM

Game UI
Est: 4h
Developer: AS

(Tasks in progress; each swimline matches one user story)

Burn Down

(current status)

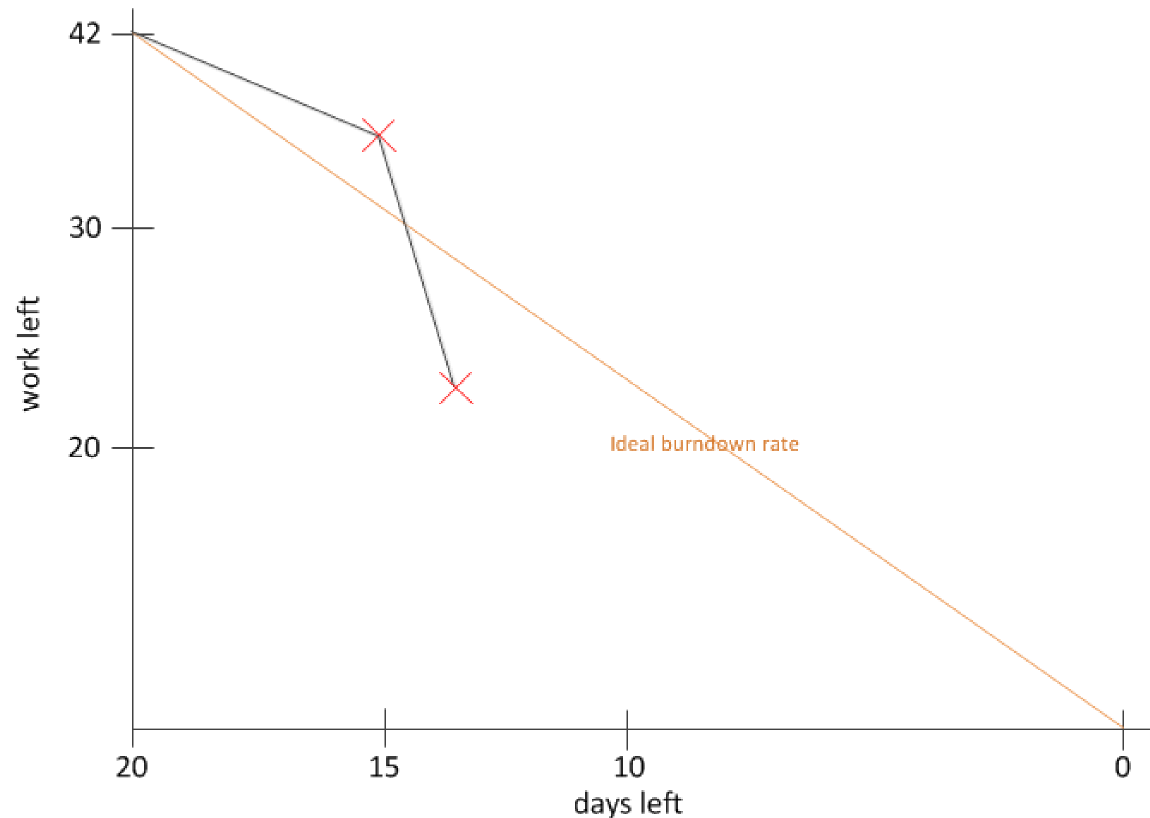
Next

(Stories moved to next iteration)

Completed

(Completed Stories)

Part IV – The Burn-Down Chart (I)



The Burn-Down-Chart shows the remaining work, NOT the actual required working time

Part IV – The Burn-Down-Chart (II)

- The chart shows
  - X-Axis: working days left until the end of the iteration
  - Y-Axis: sum of task estimates yet to be done
  - The straight line is the ideal burn-down rate:
    This is how the tasks are planned against the time available.
- During Daily Scrum Meetings, the current status is added:
  - The sum of the remaining task estimates are plotted on the intersection with remaining days.
  - If the point lies above the ideal burn down rate, the team is behind schedule. Else, it is ahead of schedule.
- The chart needs to be updated when tasks change their status, are added or removed or **when estimates change**.

Part IV – Problems

- It shows up on the chart if a **task takes longer** than expected.
  - The customer must be notified/asked for clarification.
  - The functionality must be reduced or some Sprint Backlog Items have to be scheduled for the next Sprint.
  - The reason taken into consideration for the next estimation (or velocity).
- **Unplanned Issues** like new ideas of the customer as well as bugs or other maintenance work need to be considered.
  - They become new Items with an estimate and a priority and are split into manageable tasks.
  - Items which concern the Sprint Goal are added to the Sprint Backlog – other Sprint Backlog Items may be scheduled for the next Sprint.
  - New Ideas and low priority Items are added to the Product Backlog.

Part IV – Sprint Review

- A Sprint comes to an end when time runs out.
- At this point, a **running version of the software must be available** – if not all tasks/items were handled, these have been pushed back before.

- In the Sprint Review of about four hours,
  - … a demo is given to the customer.
  - … is summarized what went wrong and what right
  - … is discussed what was achieved
- **Note**: The Sprint Review should not be extensively prepared, often PowerPoint slides are forbidden.

Part IV – Sprint Retrospective

- In the Sprint Retrospective, we want to **learn from the past:**
  - Revisit estimates – why did they differ from the actual time? What can be done better next time?
  - Calculate the new velocity, but keep in mind that the team velocity should only account for overhead, not as a buffer for wrong estimates

velocity = estimated days / required days

  - Revisit team composition, tools, methods of communication…

- **The next iteration** begins just like the last.

Part IV – Abnormal Sprint termination

- Very rarely, a Sprint must be cancelled earlier if
  - … a Sprint Goal becomes obsolete (e.g. the customer's priorities change heavily)
  - … the Sprint Goal is not achievable (e.g. the Scrum Team cannot manage the selected Backlog)
  - … too many impediments occur (e.g. the Scrum Master and the management fail in removing impediments)

- Note: Abnormal Sprint termination **consumes resources** for re-grouping and re-planning.

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Part IV – Summary

- Scrum is a **Controlled Process** to stay on top of the current progress, problems and changes.
- During a Sprint,
  - … the set of functionalities to implement does not change.
  - … the current progress is always transparent.
  - … impediments are immediately dealt with.
- Before the next Sprint,
  - … the last Sprint is reviewed to enhance productivity.
  - … priorities and estimates are re-adjusted.
  - … new ideas and functionalities are taken into consideration.

Part V/VI: Development in Detail

Part V – How to develop?

- The next section should introduce some useful means for getting hands on the software.
- We shortly revisit:
  - Testing
  - Managing Bugs
  - Productive Development in a Team
  - Software Design

Part V1 – Testing

- Testing is one of the **most important tasks** in software development.

- A test is an **executable** piece of code which **automatically executes part of the system** and **verifies** the output
  - For example, test code might start a new game and verify afterwards that is has indeed been started.
- A test may have two results:
  - **Pass (Green)**: Everything went as expected.
  - **Fail (Red)**: The system failed to meet requirements.

Part V1 – Why Testing?

- A good test leads to **confidence** in code.
- **Passing tests of a task** should mean that
  - ... new implemented functionality really works as expected.
  - ... refactored functionality or added functionality did not break any previously working code (called regression test).

- Thus, tests have to be written for, and as part of, tasks.
  - There should be a test for each important functionality realized by the task.
  - A task is **not fully implemented** if there are no associated tests.

Part V1 – How to write tests I

Ideas for writing tests:

- **Main functionality** (e.g. test that the main path works)
- **Branch-Based Testing** (e.g. check that there is a test for every branch of every condition)
- **Proper Error Handling** (e.g. check that methods correctly deal with null inputs, closed resources, failed connections)
- **Working as Documented** (e.g. if the documentation defines rules for a method, test these rules)
- **Resource Constraint Handling** (e.g. check that the system handles denied requests for resources such as database connections)

Granularity of tests:

- **Unit tests** test the smallest testable part of the software (e.g. a single method in Java).
- **Integration tests** test the interaction between components (e.g. public interfaces).
- **System tests** test the software as a whole.
- **System integration tests** test whether the software is correctly integrated into its environment.

Part V1 – Using Mocks

- Ideally, the code under test has no external dependencies.
- Unfortunately, this is mostly not the case.
  - For example, **a currency converter class** might need a database for retrieving exchange rates.
  - To test such the currency converter class, the database access object is **replaced by a mock object**.

- A mock object **mimics a real object** by implementing the same interface and just returning constant values.
- We will use Mockito for mocking purposes.
  (more details in the talk on technologies)

Part V1 – Testing in Scrum

- In Scrum, tests are an **integral part** of each Sprint – they are **NOT** deferred to the end of the project!

- Tests can be written by hand or using **Test Frameworks**.
  - The most well-known one for Java is **JUnit**.
  - The advantage is a good infrastructure and an existing test-runner with reporting functionality
    (more details in the talk on technologies)

- All tests should be **automatable**. This ensures that they can be run again and again if new functionality is added.

Part V1 – Code-and-Test

- The standard method for writing tests is **Code-and-Test**.
  - The code for the task is written.
  - Immediately afterwards, the tests for the task are written.
- This ensures that each task has tests.
- But, it also holds the danger of designing tests according to the code and not to the requirements.

- JUnit uses a **bar** for showing passed and failed tests:
  - Red Bar: At least one test failed.
  - Green Bar: All tests passed.
- The aim is to **keep the bar green**.

- In agile methods, **Test-Driven-Development** (TDD) is used.
  - The test code for the task is written.
  - Afterwards, the simplest code is written to get the test pass.
  - At last, the code is refactored.
- TDD leads to
  - … more testable code as testing drives the implementation
  - … more reasonable tests as tests are designed according to the requirements
- The aim is **red – green – refactor**. All tests fail initially, Then, the code should work and at last it is cleaned up.

Part V1 – When did we test enough?

- Testing is, in principle, a never-ending activity.
- The main criteria for moving on is **confidence**. That is
  - … the feeling that the tests adequately cover the functionality implemented in a task
  - ...or reaching a certain **code coverage** with the tests.

- **Code Test Coverage** is the percentage of code tested.
- Tools like **EclEmma** for Eclipse calculate this percentage based on the test cases.

Part V1 – Summary

- Software development does **not work** without tests!
  - Tests are **executable requirements.**
  - Tests ensure that existing functionality still works after changes (**regression testing**).

- Testing gives developers **confidence** for boldly moving forward to the next task.
- A task is implemented if the tests pass (but not yet done!)
  - See Definition of Done

Part V1 – Afterthought

Fear leads to anger, anger leads to hate, hate leads to suffering.

No tests lead to fear.

Part V2 – Managing Bugs

- It is a simple, but inevitable fact of life that **bugs happen**.
- In agile methods, **bugs are accepted like that** – nothing to be (too) ashamed of.
- A bug is therefore **treated like a normal Backlog Item**
  - A bug report is made => a new Item for the Sprint Backlog
  - The task is given an estimate and a priority (as usual).
  - It is scheduled (as usual).
- A bug task is attached to an existing User Story or a new Issue is created for it

A bug report should consist of:

- **Summary** – one sentence
- **Steps to Reproduce** – from a **well-defined** state of the system, what needs to be done to reproduce the bug?
- **What was expected, and what did happen** – to ensure everybody knows what was perceived as a problem
- **Version, Platform, Location Information** – bugs may be different in different versions, on different platforms or on different URLs
- **Severity and Priority** – how disastrous is the bug? How soon should it be fixed?

Part V2 – Testing for Bugs

- Bugs have a nasty habit of reappearing.
- Therefore,
  - Like a usual task, a bug-fixing task **MUST** include a test which reproduces the exact circumstances the bug was found in.
  - The test is added to regression testing (as usual) to ensure the bug does not occur again.
- **Finally:** When fixing a bug, look out for similar issues in the code.

Part V2 – Summary

Bugs are nothing to be ashamed of.

Bugs are treated like normal Backlog Items.

- They are written down and either attached to a User Story or a new Issue is created.
- They are estimated, prioritized, and scheduled.
- Tests are written.

Productive development in a team means

- ... using an **IDE** for managing and controlling code, dependencies and libraries
- ... using **version control** to merge the work of multiple developers in a controlled fashion
- ... using **continuous integration** for ensuring up-to-date, tested builds (manually or automated)
- ... performing **code reviews** for ensuring high code quality and bug-freedom

Part V3 – Using IDEs

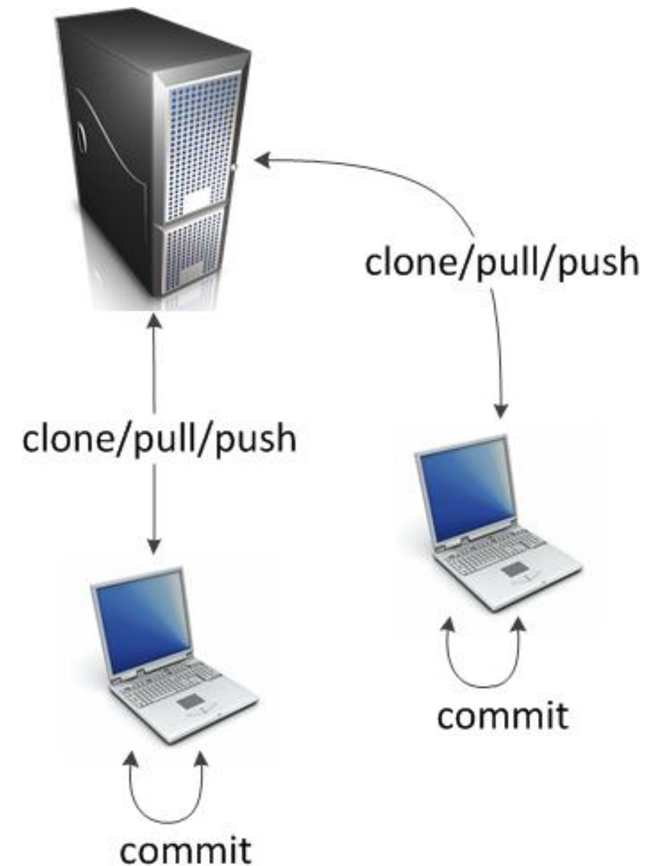
- An **integrated development environment** offers much more than just a code editor...
  - Integrated build system (background building)
  - Refactoring support (includes changing references)
  - Integrated documentation
    (source code of the entire Java API and libraries)
  - Code Navigation
    (jump to definition, references, call hierarchy, etc.)
  - Integrated test runners (JUnit and others)
  - Version Control support (CVS, Subversion...)
- An IDE makes programming **productive!**

- Problems arise when multiple developers work on the same source code:
  - Changes happen to the same file which must be merged.
  - Changes might need to be rolled back because of a faulty implementation (e.g. overridden or conflicting features)
  - Traceability is needed to be able to determine the origin of an artifact (e.g. the developer can be asked for clarification)

- **Version Control Systems** exist to address these problems
- … and even more.

Part V3 – How version control works

- We use the distributed version control system **Git** for which a client is included in Eclipse. It consists of
  - … **multiple local copies** of a repository which developers might use to work on the source code and which provide the full functionality of a revision control system
  - … often, one copy is **marked** as the official repository



- Committing a new revisions should only be done
  - ... if the code **compiles.**
  - ... **after** running all test cases
  - ... with a **commit message** which precisely says what has been changed or newly implemented (with a reference to the issue tracker task)

- **Before** pushing to the official repository, perform an update (pull) and **run all test cases again** to ensure nothing was broken.

- Eclipse already contains mechanisms for **building software**
  - This includes compiling java source code, ...
  - ... an export mechanism as an executable JAR file
  - ... and building arbitrary other elements with ant scripts.

- Ensuring that all tests pass is still the responsibility of the developer.
  - In small and simple projects, this can be done **manually**.
  - For larger, more complex projects, a dedicated system for compiling and testing might be necessary that performs **automatically regular** builds and test runs.

Part V3 – Continuous Integration

- A continuous integration system reacts to commits or on a timer and performs
  - … checking out all code
  - … building the project
  - … running all tests
- The result of the CI run (e.g. compilation or tests failed) is placed on a website or mailed to all developers.
- Well known CI tools:
  - CruiseControl (little bit old-fashioned)
  - Hudson/**Jenkins**

Part V3 – Reviewing Code I

- Peer code reviewing means getting your code checked by your peers before assuming an issue is fixed.
- Code reviews are the single biggest thing that improve code quality. The average defect detection rate is 55 – 60% (vs. 25% for Unit Testing)
- Peer code reviews entail **increased** productivity
  - Less time spent with reproducing and fixing bugs
  - Increases knowledge transfer about the code base

Pass-around Review

- The developer commits code to version control and informs the chosen reviewer via Mail or IM.
- The reviewer checks the changes, asks questions, discusses with the author, notes problems and bugs found.
- The developer responds and addresses the issues, and commits changes to version control.
- The review is completed.

Pair Programming

- Two developers collaboratively writing code
- One has the keyboard and codes – the “pilot”
- One checks code on the fly and reflects about alternative approaches – the “co-pilot”
- Roles switch constantly back and forth
- Pilot and co-pilot constantly discuss the code, and the review is performed on the fly.

Part V3 – Summary

Productive development in a team means

- … using an **IDE** for managing and controlling code, dependencies and libraries
- … using **version control** to merge the work of multiple developers in a controlled fashion
- … using **continuous integration** for ensuring up-to-date, tested builds (manually or automated)
- … performing **code reviews** for ensuring high code quality and bug-freedom

Part V4 – Good Design

- Good software design is a science of its own e.g.
  - … it must match the software type (business, embedded, ...)
  - … it must follow the company style
- **But:** There are rules which apply everywhere
  - **Visualize complicated parts**
  - **Keep it simple**
  - **Readable Code**
  - **Re-Use** (Design Patterns, Libraries)
  - **SOLID / DRY**
  - **Refactor**

Part V4 – Visualize complicated parts

- The Unified Modeling Language (UML) is a visual design tool for software.
- The static parts, in particular **class diagrams**, are a great tool for **planning** (parts of) the software.
  - **Idea:** Focus on the overall structure, not on every detail
- Diagrams also serve as **documentation** of the software for new developers.
- On a higher level of abstraction, even the customer can get some **insights into the architecture** of the software.

Part V4 –
Keep it simple

- The job of developers is implementing the task at hand … and nothing more.
- This means:

Implement the simplest thing that could possibly work!

- The aim is not to get caught up in „what might be needed in the future“.
- Instead, implement the task at hand, and implement it well.

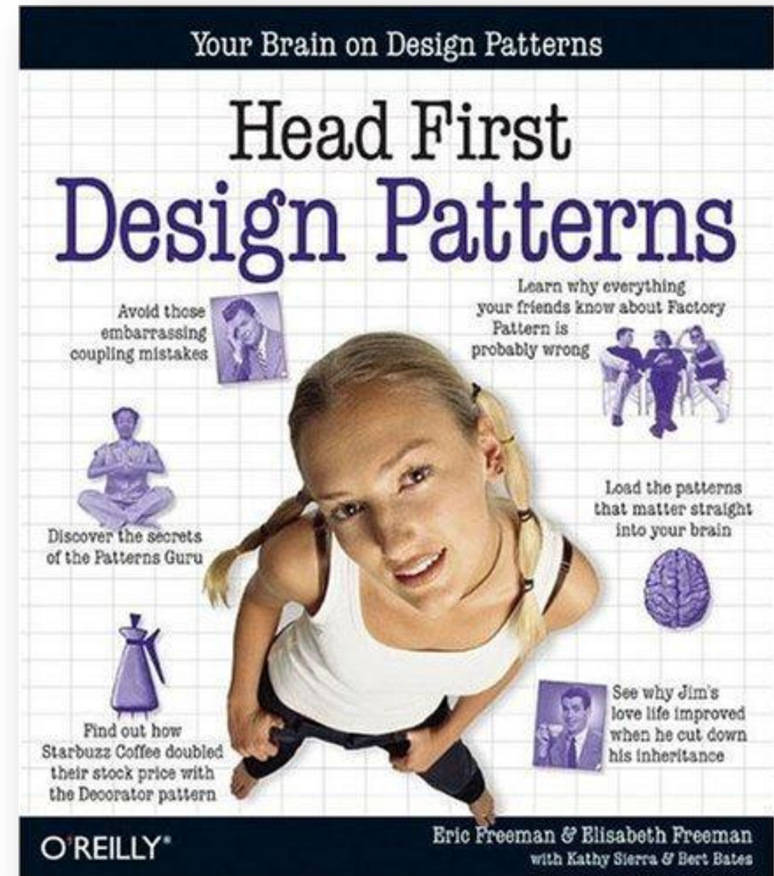
- This is (obviously) **WRONG:**

It was hard to write, it should be hard to read!

- Code should be designed to be **easy to read.**
  - “Speaking“ and “Readable” Code:
    - Use long, **self-explanatory** variable and method names.
    - Use the formatter to ensure everything looks the same.
    - Prefer code to documentation.
  - **But:** Use JavaDoc if the code contains pitfalls
    - i.e. it is not obvious why it was written this way

Part V4 – Re-Use

- Do not reinvent the wheel!
- Mostly, there are already solutions for your problems:
  - Check for applicable **design patterns**
  - Check the (Java) **API**
  - Check for **external libraries**
- **Talk to your team!**



Maintaining the code is easier as one only has to look in one place

- **Single Responsibility Principle**
  - If a task is split across several classes, all of them need to change if the task changes.
  - **Result**: maintenance nightmare
- **Solution**: only **one responsibility** per class
  - Aim: high cohesion and low coupling

- **Don‘t Repeat Yourself**
  - If a bug is found in copied code, it needs to be changed **everywhere.**
  - **Result:** maintenance nightmare (again)
- **Solution**: Use inheritance/delegation to pull out common code
  - Aim: Find generic functionality (Hint: copy&pasted code)

Part V4 – Refactor

- One of the best things about IDEs is **refactoring support.**
- Due to design purposes, code may change:
  - Elements change their meaning.
  - Elements have to be moved.
  - Elements have to be split or merged.

- **Never** refrain from restructuring and renaming your code to fit the current view of the system.
  - Refactoring take care of all references automatically.
  - The aim is having **no burdens of the past.**
    (“this field is called xy because, at the beginning, we thought...“)
  - And **the tests ensure that the code still works**.

Part V4 – Summary

- Visualizing, Creating simple and readable Code, Re-Using, SOLID/DRY, and Refactoring are tools waiting to be applied.

- But: Do not go too far!
  - Even a "Perfect Design" is obsolete tomorrow.
  - Aim for "**good-enough design**".
  - Unfortunately, only experience helps to find the right balance.

Part VI/VI: Scaling Scrum

Part VI – More than 9 Developers

- Sometimes, we need more than 9 developers:
  - Time constraints
  - Scope of project
- HOWEVER:
  > 9 developers in one Scrum Team induce complex coordination

How do we scale Scrum to larger projects?

Part VI – Several Scrum Teams

To scale Scrum, we implement several Scrum Teams:

- Each Scrum Team is an own unit which means…
  - … it has its own Sprint Goal (and therefore Sprint Backlog).
  - … it has its own team dynamics.
- The Scrum Teams collaborate in the same project by…
  - … working on the same Product Backlog.
  - … sharing their progress in Scrum of Scrums Meetings.

Part VI – How to extend Scrum (I)

- Sprint Planning Meeting
  - Each Scrum Team gets its **own Sprint Goal**.
  - Each Scrum Team (one after another) selects its **own Sprint Backlog** according to its Spring Goal.
- Daily Scrum Meeting
  - Each Scrum Team organizes its **own Daily Scrum Meeting** where all other Scrums Teams act as chickens.
- **Scrum of Scrums Meeting**
  - Each Scrum Team designates **two representatives**.
  - Representatives of every Scrum Team meet to discuss their work **focusing on areas of overlap and integration**.

- Sprint Review
  - Each Scrum Team presents its results **independently**.
  - The results are presented **integrated** into the whole product (and not separated from the results of other Scrum Teams).

- Sprint Retrospective
  - Each Scrum Team reviews the last Sprint **independently**.

Part VI – Summary

To scale Scrum, we implement several Scrum Teams where

- … each Scrum Team pursues its own Sprint Goal,
- … but all Scrum Teams collaborate in completing the same project vision

Minimize interactions and dependencies between teams!

Maximize cohesion within each team!

Conclusion

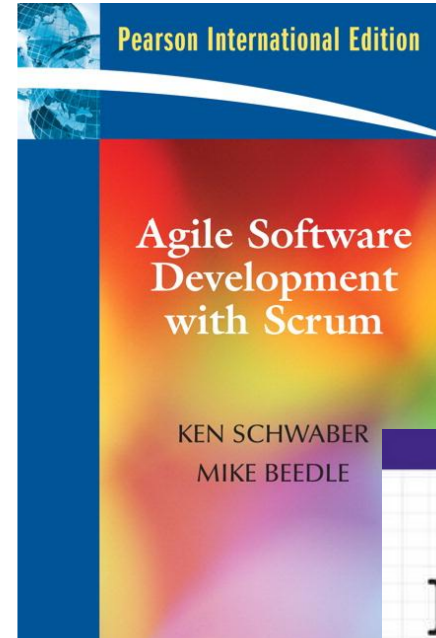
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Values in Scrum

- Commitment
  - The Scrum Team has full authority how to do the work.
  - The whole Scrum Team is responsible for the whole product.
- Openness
  - Everything is visible to everyone.
- Courage
  - Do your best, don’t give up!
- Respect
  - Respect everyone’s strengths and weaknesses!
  - Provide help and do your best!

The Scrum process for SWEP

- This talk has presented an agile method based on Scrum, XP and the HFSD process.
- Please make yourself familiar with the process in the reminder of the week!



+ scrum.org