Agile Software Development (DIT191 / EDA397)

Lecture 3: Agile methods crash course

Eric Knauss
<eric.knauss@cse.gu.se>

Organisatorial

- Teams have been assigned and are online
- Guest lecture by Thomas Lüvo confirmed “Scaling agile”
- Hunt for larger rooms ongoing

Course Objectives

Knowledge and understanding	Skills and ability	Judgement and approach
Compare agile and traditional softw. dev,	Forming a team organically	Explain: people/commun. centric dev.
Relate lean and agile development	Collaborate in small software dev. teams	Apply fact: people drive project success
Contrast different agile methodologies	Interact and show progress continuously	Describe: No single methodology fits all
Use the agile manifest and its accompanying principles	Develop SW using small and frequent iterations	Discuss: methodology needs to adopt to culture
Discuss what is different when leading an agile team	Use test-driven dev. and automated tests	
	Refactor a program/design	
	Be member of agile team	
	Incremental planning using user stories	

Sprint 1

Sprint 2

Sprint 3

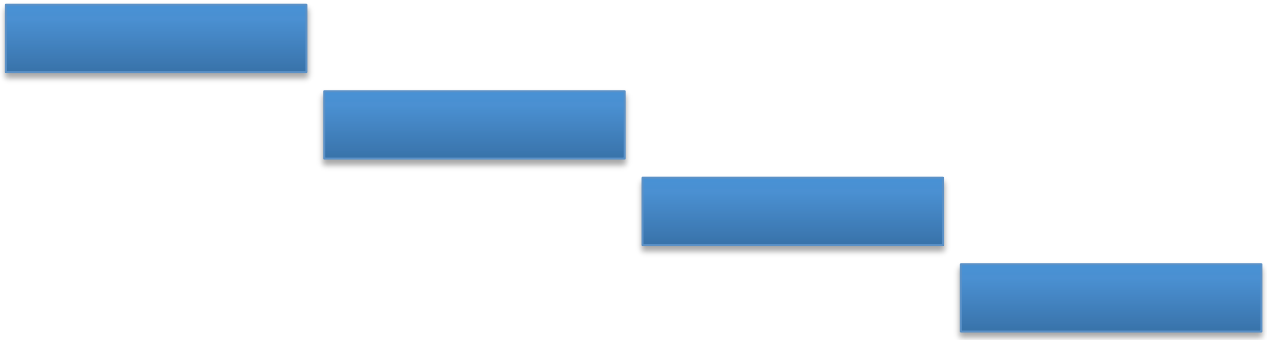
Examination

- Written exam, individual, 3.0 credits
  - 60 points, 24 required to pass
- Project, 4.5 credits
  - Grades: Fail/Pass but gives bonus on written exam
- Grades
  - Chalmers:
    - [0-49%] → Fail,
    - [50-64%] → 3,
    - [65-79%] → 4,
    - [80-100%] → 5
  - GU
    - [0-49%] → Fail,
    - [50-79%] → G,
    - [80-100%] → VG

Where are we coming from?

Systematic sequential development

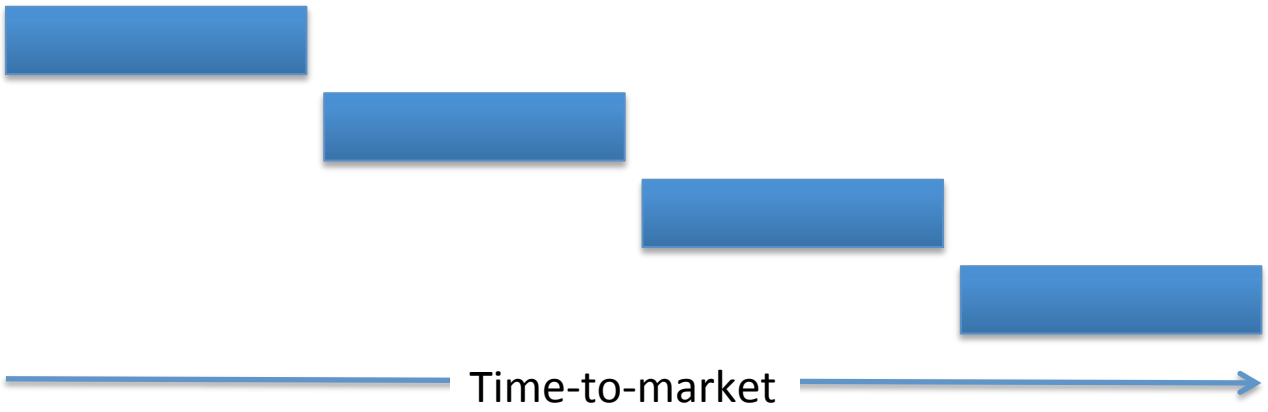
- Requirements
- Design
- Programming
- Test



- Advantages
  - Simple
  - Controllable
  - Cost efficient
- Problems
  - Time-to-market
  - What about change?

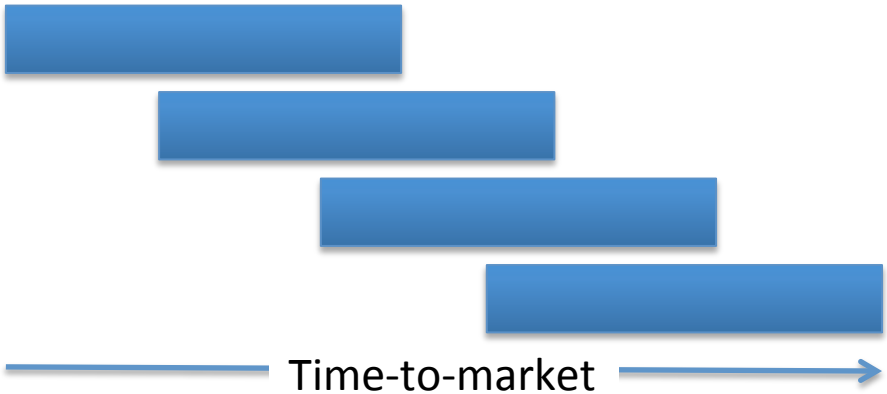
Towards concurrent development

- Requirements
- Design
- Programming
- Test



What can we do if time to market and robustness against late changes are more important than cost-efficiency?

- Requirements
- Design
- Programming
- Test



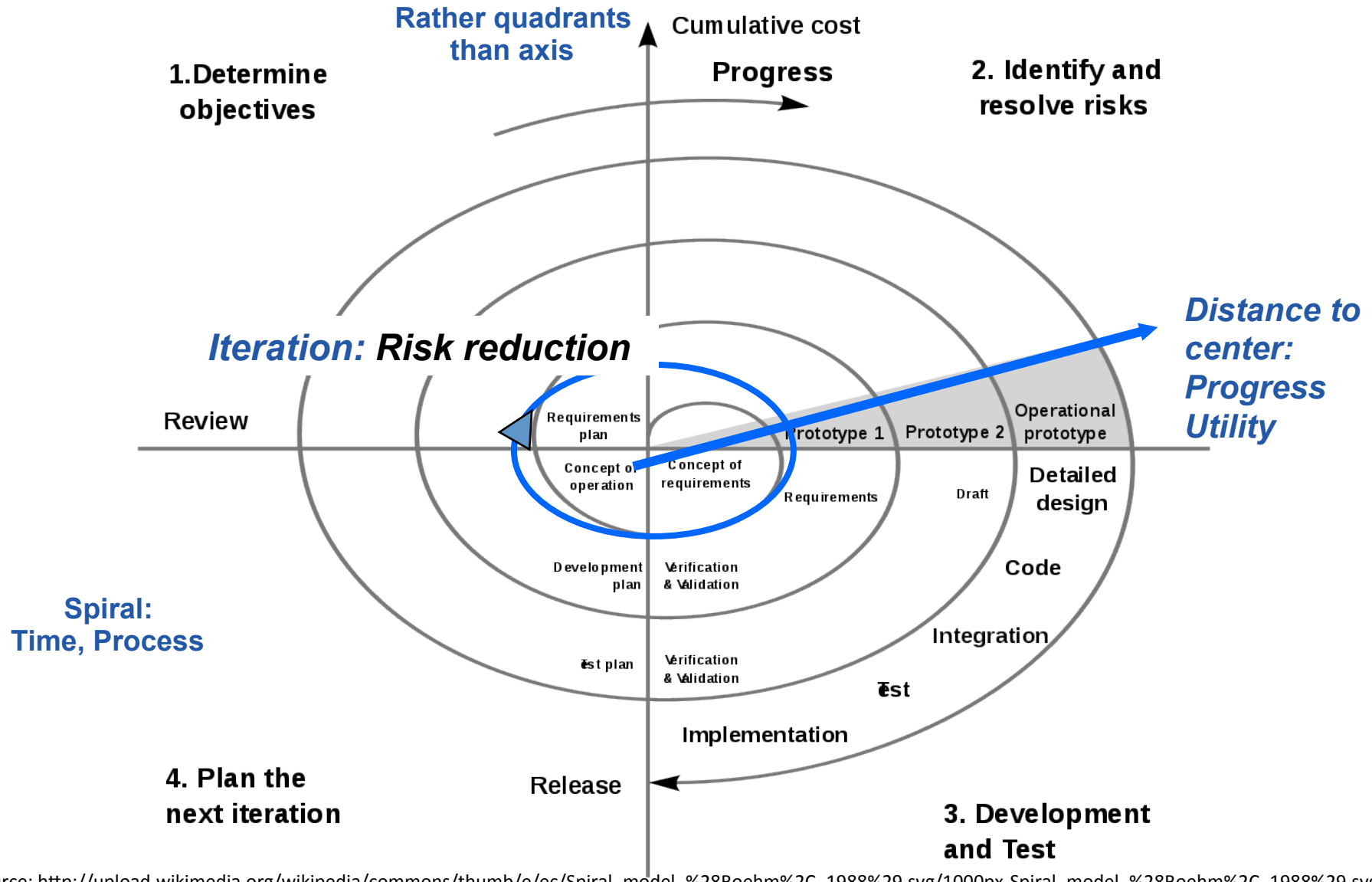
Iteration Models

- System requirements *always* evolve during a project
- Iterations are part of larger development projects
- Iterations can be applied to any generic development process model

Incremental delivery

- Customer value can be delivered with each increment so system functionality is early available for customer's feedback
- Early increments act as prototype to help elicit requirements for later increments
- Reduced risk of project failure
- The highest priority system services tend to receive the most testing

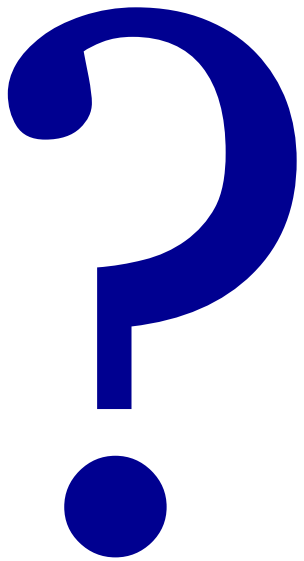
Spiral Model [Boehm]



Source: http://upload.wikimedia.org/wikipedia/commons/thumb/e/ec/Spiral_model_%28Boehm%2C_1988%29.svg/1000px-Spiral_model_%28Boehm%2C_1988%29.svg.png

Spiral development

- Objective setting
  - Specific objectives for the phase are identified
- Risk assessment and reduction
  - Risks are assessed and activities put in place to reduce the key risks
- Development and validation
  - A development model for the system is chosen which can be any of the generic models
- Planning
  - The project is reviewed and the next phase of the spiral is planned



- Is the Spiral model agile?
- What kind of prototypes will be developed in the Spiral model?
- Is consecutive prototyping agile?

What is a prototype?

Why was it constructed?

Horizontal
Vertical

Explorative
Experimental
Evolutionary

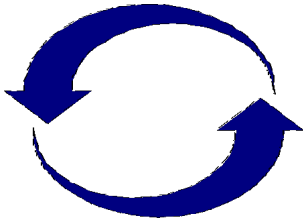
Presentation- PT
Actual PT
Laboratory sample
Pilot system

How do end-users/colleagues/you like the PT?
What was learnt or experienced during construction / presentation of prototype?

+/-

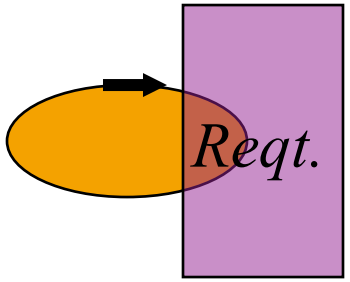
What is the contribution of the prototype?
What is (only) infrastructure?

Technical knowledge during the process that can be valuable somewhere else

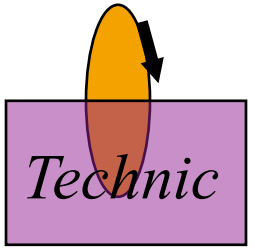


Prototyping approaches

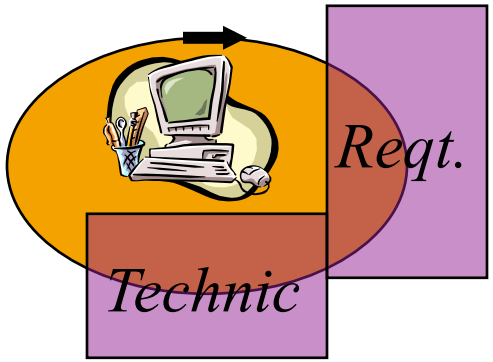
- Explorative
  - Goal: Identify requirements
  - Show alternatives, capture feedback



- Experimental
  - Goal: Assess technical alternatives
  - Test feasibility and implementation



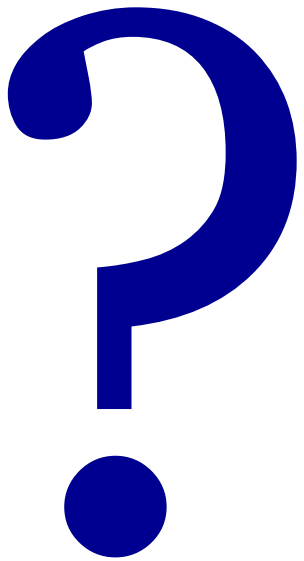
- Evolutionary
  - Goal: Fitting system despite changing requirements
  - Adjust running system constantly to changing requiremetns and constraints



Evolutionary Development

- Applicability
  - Small to midsize projects
  - Parts of larger systems (e.g. user interface)
  - Short lifetime projects

- Restrictions
  - Process visibility for the customer (what is finally delivered?)
  - Risk that systems are only poorly structured
  - Special skills in rapid prototyping languages are necessary

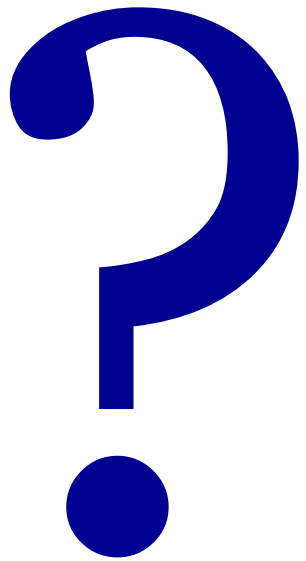


- What is your position about the following statements:
  - “When a customer accepts our prototype, we polish it a bit more and deliver it as product.”
  - “We write our prototypes in a different programming language to discourage copy&paste to the main product.”

Agile Manifesto

http://agilemanifesto.org

- Began as a provocation: Plan-driven development did not safe the Software world…
- Now a very serious movement, well adapted in industry.
- There are a couple of established agile methods: How to integrate these values in everyday software development



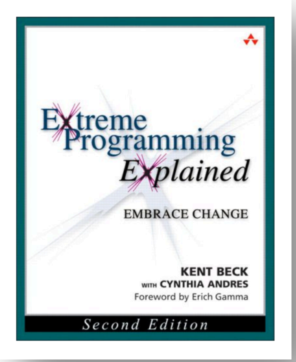
- Can the following projects be agile?
  - App development
  - Online shop
  - Mission controller for Airplane
  - Controller for nuclear plant

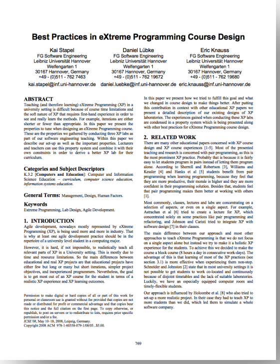
Agile Methods: eXtreme Programming (XP)

- Extreme programming:
  - An approach based on the development and delivery of very small increments of functionality
  - No fine grained process description, but 12 practices arranged around short development circles (4-6 weeks)
  - “Turn-to-ten” metaphor (refers to volume setting of older amplifiers):
    - Reviewing is good? → Review continuously: Pair Programming
    - Early Tests are good? → Write tests before code: Test-First
    - Customer interaction is good? → Have Onsite-Customer
    - …

Planning Game

- Business people need to decide
  - Scope, Priority, Composition of release, dates of release
- Technical people need to decide
  - Estimates, Consequences, Process, Detailed scheduling




Best Practices in eXtreme Programming Course Design

Kai Stapel, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 7463, kai.stapel@inf.uni-hannover.de
Daniel Lübke, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 19672, daniel.luebke@inf.uni-hannover.de
Eric Knauss, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 19680, eric.knauss@inf.uni-hannover.de

ABSTRACT
Teaching (and therefore learning) eXtreme Programming (XP) in a university setting is difficult because of course time limitations and the soft nature of XP that requires first-hand experience in order to see and really learn the methods. For example, iterations are either shorter or fewer than appropriate. In this paper we present the properties to tune when designing an eXtreme Programming course. These are the properties we gathered by conducting three XP labs as part of our software engineering teaching. Within this paper we describe our set-up as well as the important properties. Lecturers and teachers can use this property system and combine it with their own constraints in order to derive a better XP lab for their curriculum.

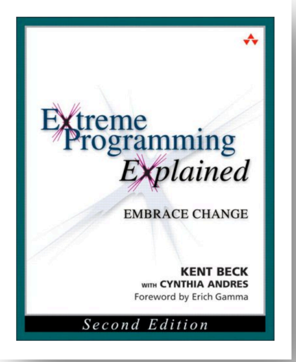
In this paper we present how we tried to fulfill this goal and what we changed in course design to make things better. ...

“Students learn how to divide requirements into User Stories and how to prioritize and estimate the costs of these stories. While such tasks seem to be easy in theory, dealing with dependencies in the planning game is normally a challenge for inexperienced developers like students.”

- (+) More iterations, small teams, customer interested, technical support, progress feedback
- (-) Longer iterations

Small releases

- Every release
  - … should be as small as possible
  - … should contain the most valuable business requirements
  - … has to make sense as a whole
  - … should be delivered every 4-8 weeks (rather than 6-12 month)

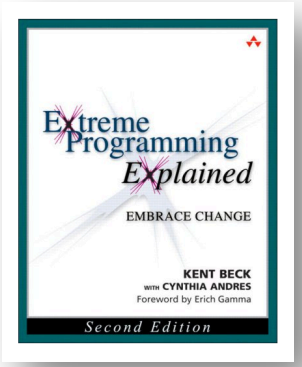


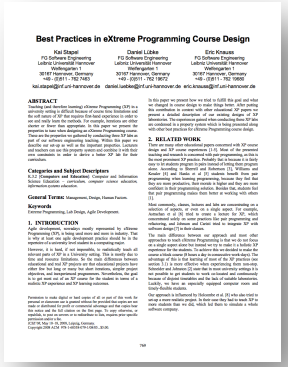
"Students learn the benefits of small releases that already offer value to the customer and how to technically put a system into production including packaging."

- (+) More iterations, progress feedback

Metaphor

- Examples
  - Naïve: “Contract management system deals with contracts, customers, and endorsements”
  - “Computer should appear as a desktop”
  - “Pension calculation is a spreadsheet”
  - Align team thinking





Best Practices in eXtreme Programming Course Design

Kai Stapel
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 7483
kai.stapel@inf.uni-hannover.de

Daniel Lübke
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 19872
daniel.luebke@inf.uni-hannover.de

Eric Knauss
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 19868
eric.knauss@inf.uni-hannover.de

ABSTRACT

Categories and Subject Descriptors

General Terms

Keywords

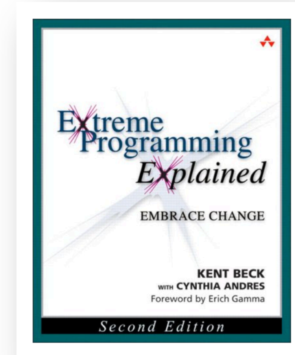
1. INTRODUCTION

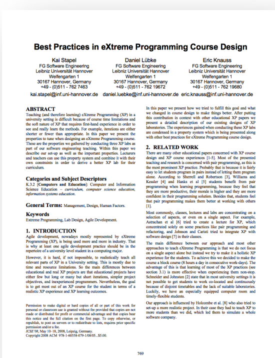
“Students learn how to develop a metaphor that helps every team member to better understand how the whole system works.”

- (+) Technical support, technical feedback

Simple Design

- The right design at any given time
  - Runs all the tests
  - Has no duplicated logic
  - States every intention important to the programmers
  - Has the fewest possible classes and methods





Best Practices in eXtreme Programming Course Design

Kai Stapel, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 7463, kai.stapel@inf.uni-hannover.de

Daniel Lübke, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 19672, daniel.luebke@inf.uni-hannover.de

Eric Knauss, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 19680, eric.knauss@inf.uni-hannover.de

ABSTRACT

Teaching (and therefore learning) eXtreme Programming (XP) in a university setting is difficult because of course time limitations and the soft nature of XP that requires first-hand experience in order to see and really learn the methods. For example, iterations are either shorter or fewer than appropriate. In this paper we present the properties to tune when designing an eXtreme Programming course. These are the properties we gathered by conducting three XP labs as part of our software engineering teaching. Within this paper we describe our set-up as well as the important properties. Lecturers and teachers can use this property system and combine it with their own constraints in order to derive a better XP lab for their curriculum.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *curriculum, computer science education, information systems education.*

General Terms: Management, Design, Human Factors.

Keywords

Extreme Programming, Lab Design, Agile Development.

1. INTRODUCTION

Agile development, nowadays mostly represented by eXtreme Programming (XP), is being used more and more in industry. That is why at least one agile development practice should be in the repertoire of a university level student in a computing major.

However, it is hard, if not impossible, to realistically teach all relevant parts of XP in a University setting. This is mostly due to time and resource limitations. So the main differences between educational and real XP projects are that educational projects have either few but long or many but short iterations, simpler project objectives, and inexperienced programmers. Nevertheless, the goal is to get most out of an XP course for the student in terms of a realistic XP experience and XP learning outcomes.

In this paper we present how we tried to fulfill this goal and what we changed in course design to make things better. After putting this contribution in context with other educational XP papers we present a detailed description of our existing designs of XP laboratories. The experiences gained when conducting these XP labs are condensed in a property system which is being presented along with other best practices for eXtreme Programming course design.

2. RELATED WORK

There are many other educational papers concerned with XP course design and XP course experiences [1-5]. Most of the presented teaching and research is concerned with pair programming, as this is the most prominent XP practice. Probably that is because it is fairly easy to let students program in pairs instead of letting them program alone. According to Sherrell and Robertson [3], Williams and Kessler [4] and Hanks et al [5] students benefit from pair programming when learning programming, because they feel that they are more productive, their morale is higher and they are more confident in their programming solution. Besides that, students feel that pair programming makes them better at working with others [1].

Most commonly, classes, lectures and labs are concentrating on a selection of aspects, or even on a single aspect. For example, Astrachan et al [6] tried to create a lecture for XP, which concentrated solely on some practices like pair programming and refactoring, and Johnson and Caristi tried to integrate XP with software design [7] in their classes.

The main difference between our approach and most other approaches to teach eXtreme Programming is that we do not focus on a single aspect alone but instead we try to make it a holistic XP experience for the students. To achieve this we decided to make the course a block course (8 hours a day in consecutive work-days). The advantage of this is that learning of most of the XP practices (see section 3.1) is more effective when experiencing them non-stop. Schneider and Johnston [2] state that in most university settings it is not possible to get students to work co-located and continuously because of disjoint timetables and the lack of suitable laboratories. Luckily, we have an especially equipped computer room and timely-flexible students.

Our approach is influenced by Holcombe et al. [8] who also tried to set-up a more realistic project. In their case they had to teach XP to more students than we did, which led them to simulate a whole software company.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

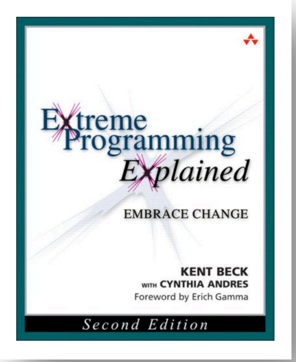
769

"Students learn the benefits of simple software design which improves their ability to change the system quickly and accommodate it to changing requirements."

- (+) More iterations, technical support

Testing

- Any feature without automated test does not exist
  - Don’t write a test for every method
  - Write a test for every productive method that could possibly break
  - “Program becomes more and more confident over time”




Best Practices in eXtreme Programming Course Design

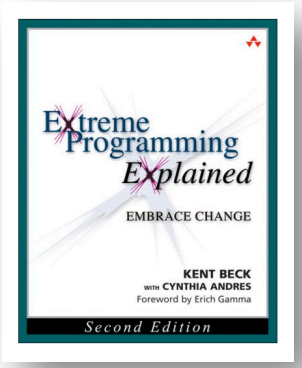
Kai Stapel, Daniel Lübke, Eric Knauss

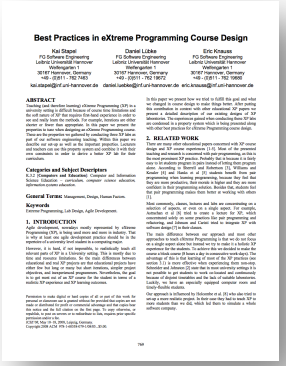
“Students learn how to use unit test frameworks and the test-first approach to build high quality software and to recognize the advantages of well-tested code when making changes.”

- (+) More iterations, **technical support, technical feedback**

Refactoring

- Is there a way of changing the program to make it easier to add a new feature?
- After adding the feature: Can we simplify the design?
- Important investment!!!





Best Practices in eXtreme Programming Course Design

Kai Stapel
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 7483
kai.stapel@inf.uni-hannover.de

Daniel Lübke
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 19872
daniel.luebke@inf.uni-hannover.de

Eric Knauss
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 19868
eric.knauss@inf.uni-hannover.de

ABSTRACT

Categories and Subject Descriptors

General Terms

Keywords

1. INTRODUCTION

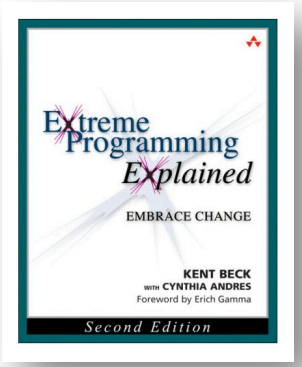
2. RELATED WORK

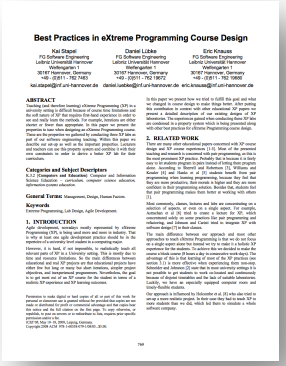
"Students learn to refactor the software to remove duplication, improve communication and simplify the code base. Especially refactoring large systems can be troublesome and is a worthy experience that can only be made in long lasting projects."

- (+) More iterations, customer interest, technical support

Refactoring

- Is there a way of changing the program to make it easier to add a new feature?
- After adding the feature: Can we simplify the design?
- Important investment!!!





Best Practices in eXtreme Programming Course Design

Kai Stapel
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 7483
kai.stapel@inf.uni-hannover.de

Daniel Lübke
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 19872
daniel.luebke@inf.uni-hannover.de

Eric Knauss
FG Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover, Germany
+49 - (0)511 - 762 19868
eric.knauss@inf.uni-hannover.de

ABSTRACT

Categories and Subject Descriptors

General Terms

Keywords

1. INTRODUCTION

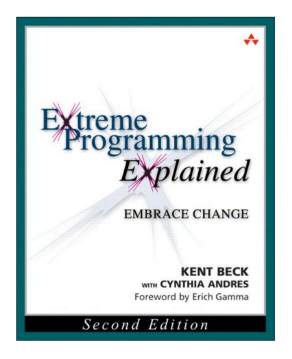
2. RELATED WORK

"Students learn to refactor the software to remove duplication, improve communication and simplify the code base. Especially refactoring large systems can be troublesome and is a worthy experience that can only be made in long lasting projects."

- (+) More iterations, customer interest, technical support

Pair Programming

- Driver: Has the keyboard, writes the code
- Navigator: Thinking strategically
  - Will this work? Which test cases might not work? Can we simplify?




Best Practices in eXtreme Programming Course Design

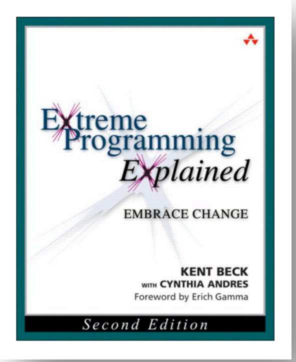
Kai Stapel, Daniel Lübke, Eric Knauss

"Students learn and experience the principles of pair programming, the advantages of writing software with a partner and the social challenges that are associated."

- (+) Block course, **small team size**, technical support
- (-) Long iterations

Collective Code Ownership

- "Anybody who sees an opportunity to add value to any portion of the code is required to do so at any time."
  - No code ownership → Chaos
  - Individual code ownership → Stable but slow



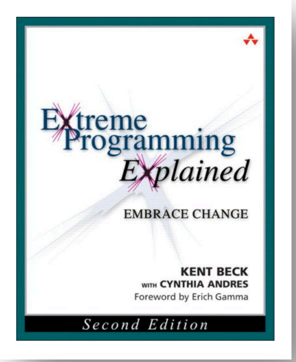

Best Practices in eXtreme Programming Course Design

"Students get to know the advantages of collective code ownership and the challenges that arise with parallel updates and changes to their own code by other team members."

- (+) Block course, small team size

Continuous Integration

- Integrate and test code every few hours (1 day at the most)
- Dedicated machine helps
  - If Machine is free: pair sits down, integrates their changes, tests, and does not leave before 100% of tests run




Best Practices in eXtreme Programming Course Design

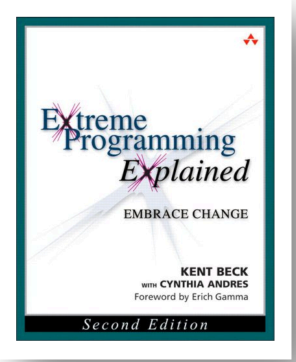
Kai Stapel, Daniel Lübke, Eric Knauss

"To counter conflicting updates to the code base, students learn to integrate and build the software frequently. "

- (+) Block course, longer iterations, more iterations, small team size
- (discovered later: technical feedback)

Sustainable pace (aka 40h week)

- Be fresh every morning, tired every night
- One week of overtime must not be followed by another one

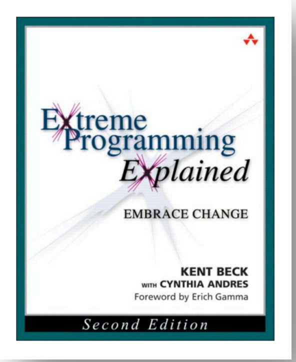


"In contrast to normal life in university, students experience to work continuously for 40-hours per week in a designated team room."

- (+) **Block course**

On-Site Customer

- Real customer in the room
  - Answers all questions now (…and can revise answer later)
  - Customer proxy
- *Answer now* more important than *answer correct*



Best Practices in eXtreme Programming Course Design

Kai Stapel, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 7463, kai.stapel@inf.uni-hannover.de

Daniel Lübke, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 19672, daniel.luebke@inf.uni-hannover.de

Eric Knauss, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany, +49 - (0)511 - 762 19680, eric.knauss@inf.uni-hannover.de

ABSTRACT

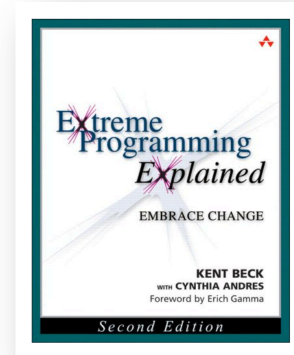
Teaching (and therefore learning) eXtreme Programming (XP) in a university setting is difficult because of course time limitations and the soft nature of XP that requires first-hand experience in order to see and really learn the methods. For example, iterations are either shorter or fewer than appropriate. In this paper we present the properties to tune when designing an eXtreme Programming course. These are the properties we gathered by conducting three XP labs as part of our software engineering teaching. Within this paper we describe our set-up as well as the important properties. Lecturers and teachers can use this property system and combine it with their own constraints in order to derive a better XP lab for their curriculum.

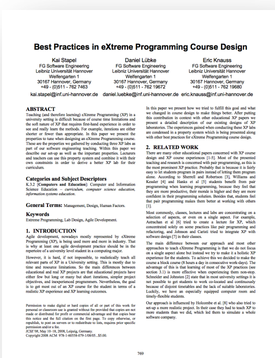
“Students have to interact with a designated On-Site Customer who is available full-time to answer questions.”

- (+) Block course, customer interest

Coding Standards

- Swapping partners, changing concurrently all parts of the code…
- Your code should better look consistently!
  - Once and only once rule
  - Emphasize communication
  - Adopted by whole team




Best Practices in eXtreme Programming Course Design

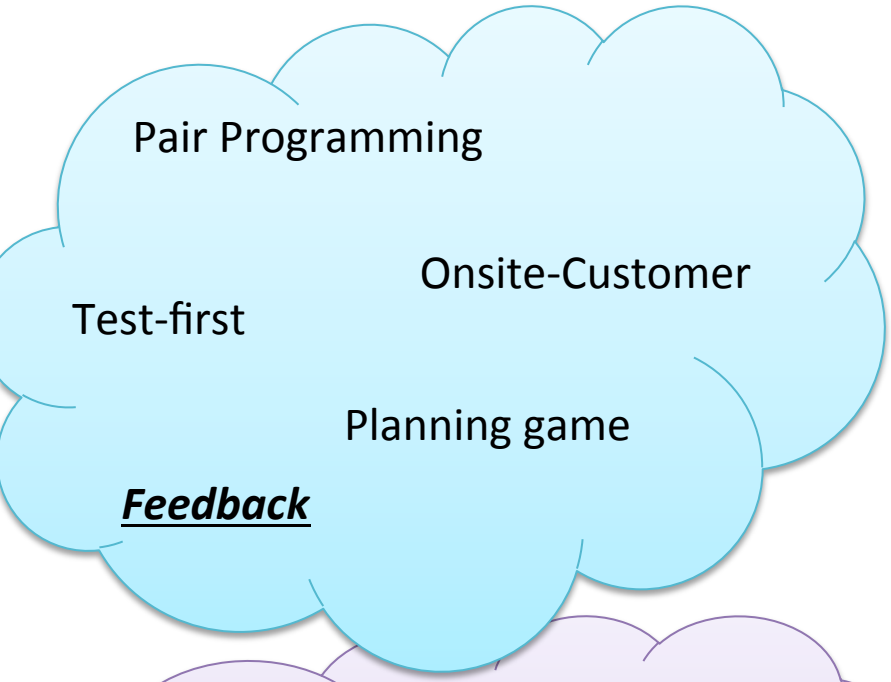
Kai Stapel, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany

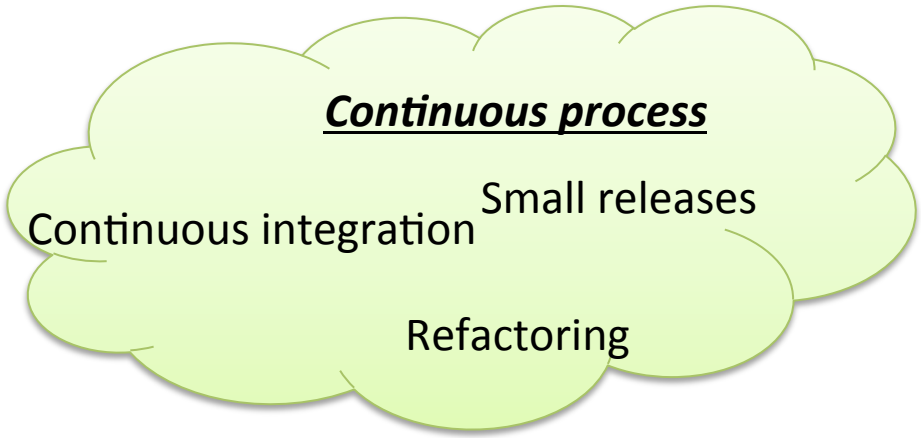
Daniel Lübke, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany

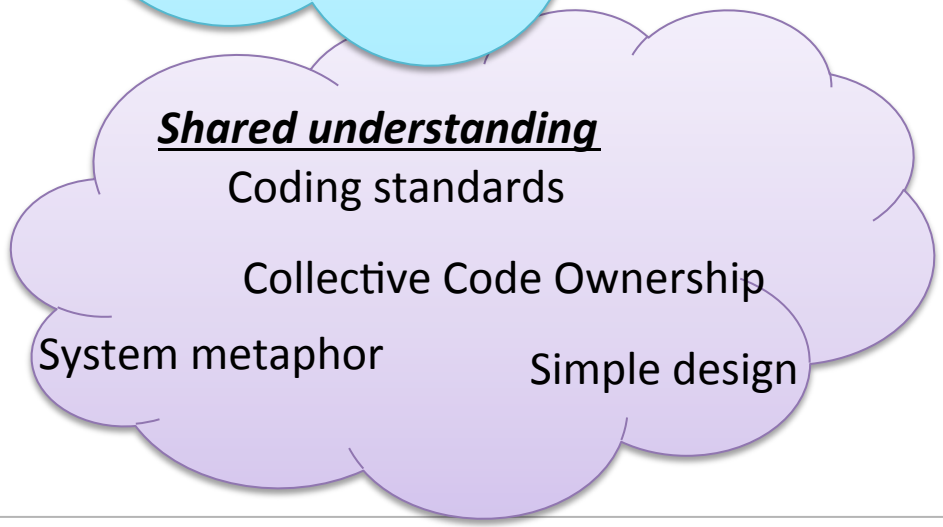
Eric Knauss, FG Software Engineering, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover, Germany

"Students experience the importance of uniform coding conventions throughout the team especially when combined with collective code ownership. "

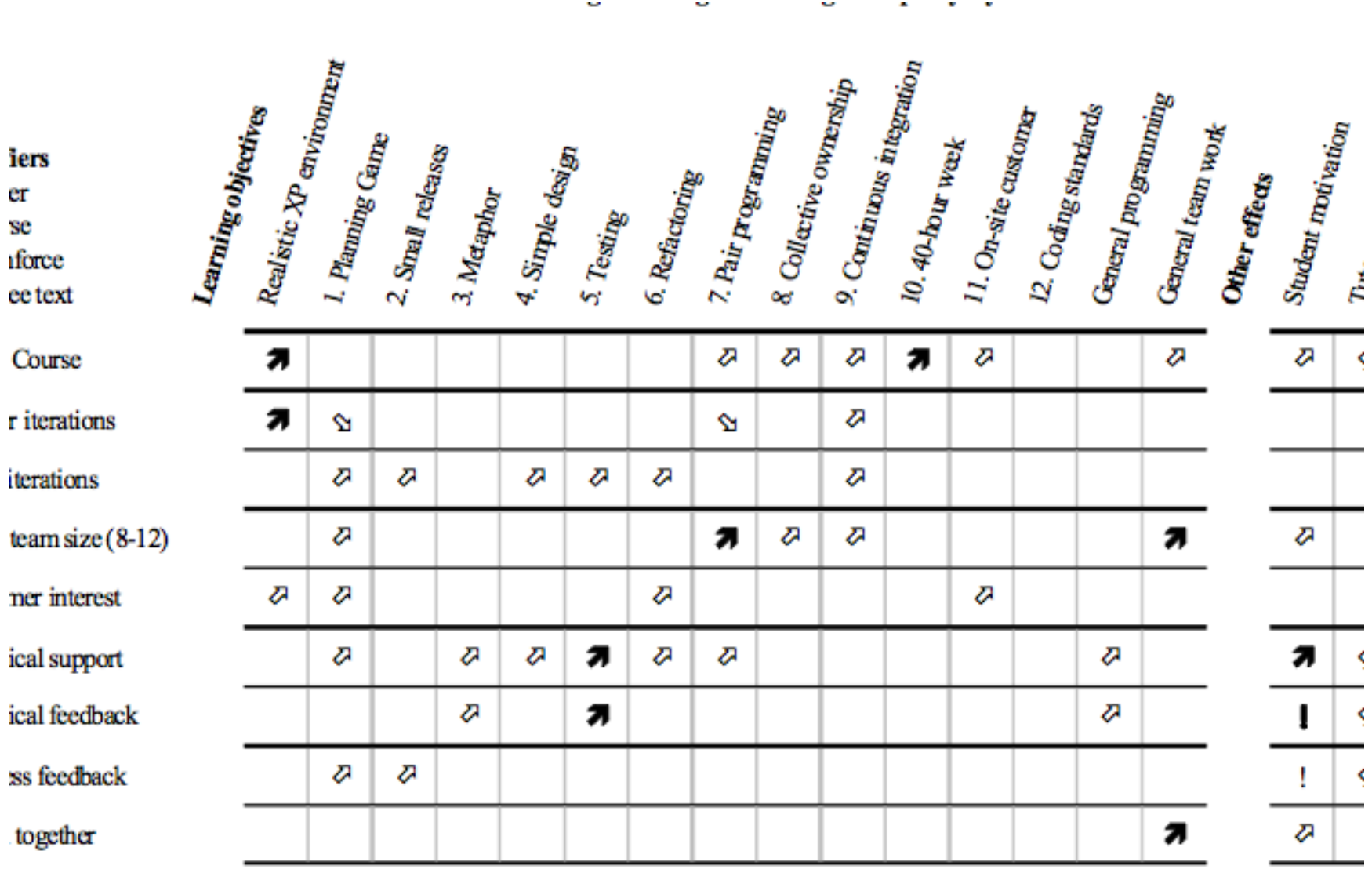
XP Practices



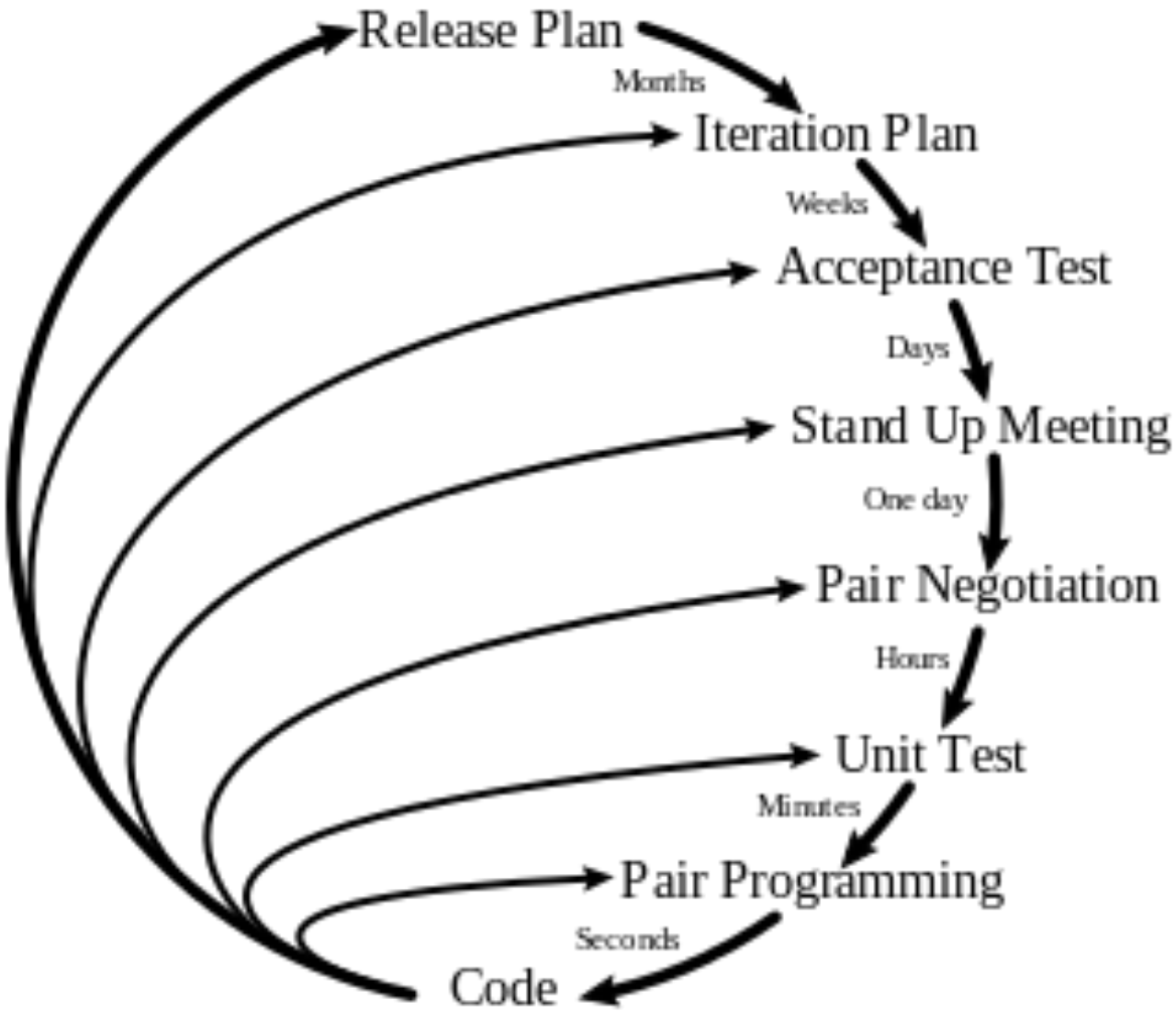




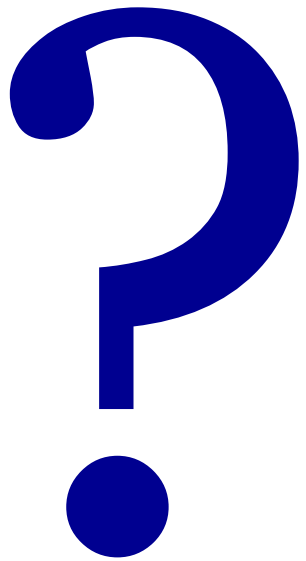
Overview: XP in Teaching



XP: Planning/Feedback Loops



http://en.wikipedia.org/wiki/File:Extreme_Programming.svg



- Why is it not a good idea to show XP as an activity diagram?
- Why do people want to do that?

SCRUM Approach

- Basic idea:
  - Bundle requirements, do not forward continuous changes of requirements to team
  - Changes: often and appreciated – but put a Baseline in between
  - SCRUM-Master is a buffer between interal and external stakeholders
  - Daily meetings facilitate direct communication
- After initial phase: Project flows
  - Self-facilitating on informational and psychological level
- SCRUM focusses on essential aspects
  - Everything not essential can be decided by the team

SCRUM Practices

Hints

- SCRUM Meetings
  - SCRUM Master minds the time (2-3 min./Person)
    - Standup-meeting: faster
    - Replace status meetings – safe time
  - Important: Always same time and place!
    - (not important: where)
    - Daily / frequent meetings avoid long/quiet crisis
  - Content
    - What was done since the last meeting?
    - What is planned to be done before the next meeting?
    - Found obstacles? Write on whiteboard!
  - Useful: Share information and facilitate social aspects
  - Schedule further meetings to follow up on things (e.g. obstacles)

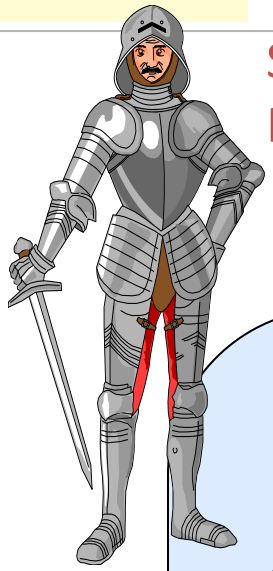
SCRUM Practices

Hints

- Sprint
  - During sprint: Autonomous team: *Pioneers*
    - No new requirements / no changes
    - No external influences
    - Only sprint goal
      - Fixed: Time (approx. 30 Tage), Cost (Developers etc.), Quality
      - Variable: Functionality
        - » Team can adjust details and scope of functionality based on the time-cost-quality frame and with respect to the sprint goal
  - Sprint can be cancelled
  - After Sprint: 4h Sprint-Meeting
    - Avoid long preparation (max. 2h)
    - Avoid slides
    - Often very informal
- Adjust SCRUM (longer Sprints, other Meetings…)
  - Okay, after being successful with the traditional setup
  - Only based on experiences – never without experience

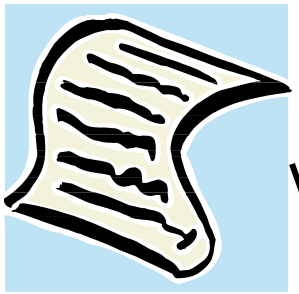
Scrum

Others



SCRUM Master

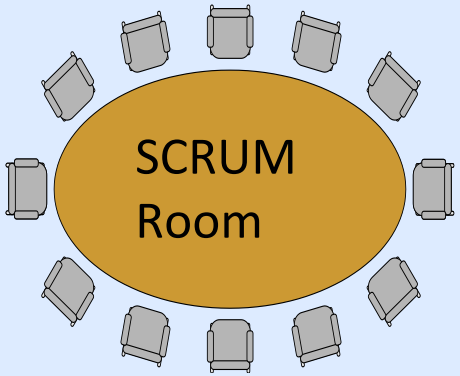
Product Owner



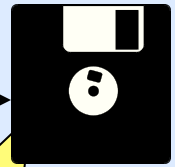
Product Backlog (prioritized)

SPRINT Backlog

SCRUM team 7+/-2



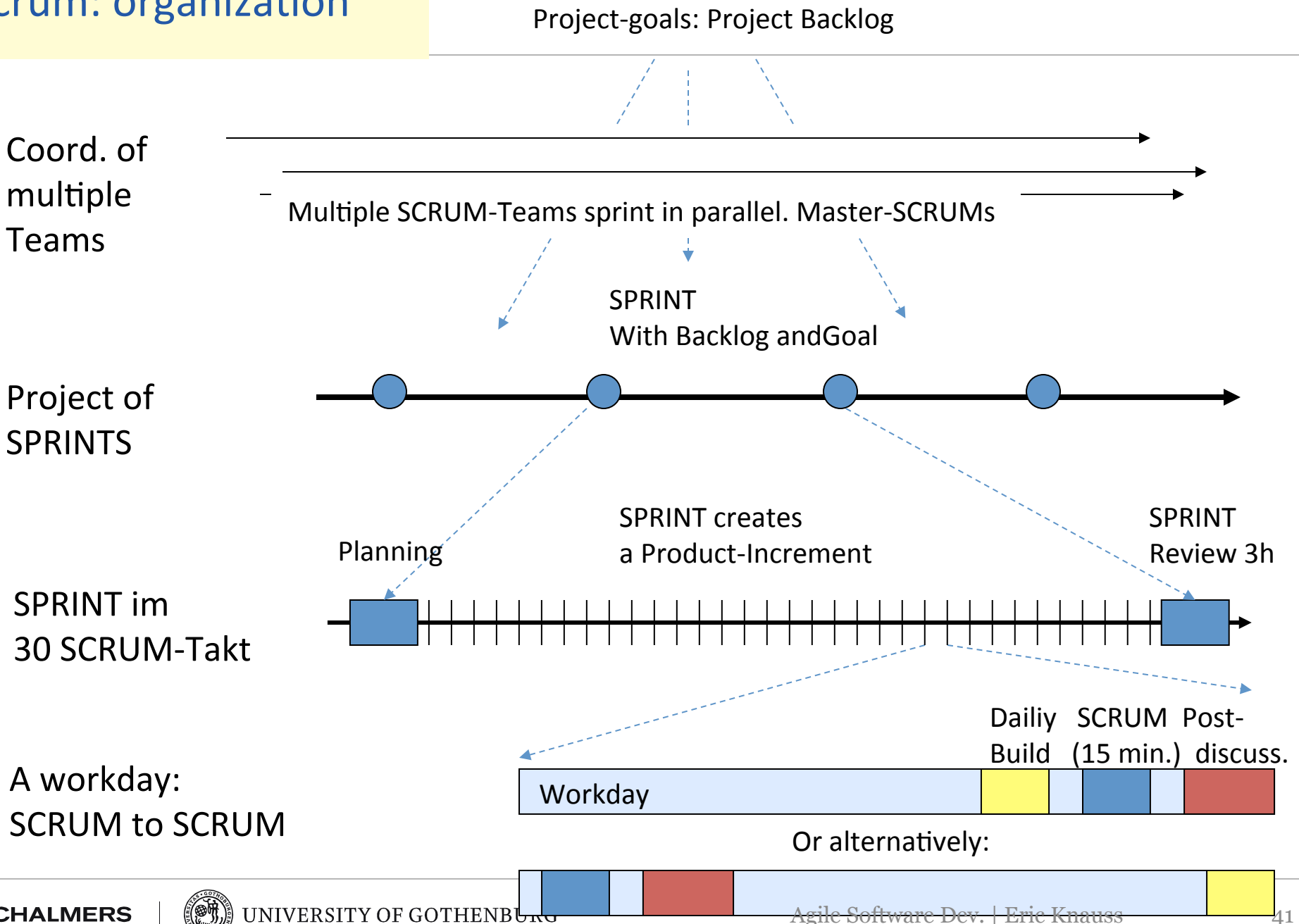
Increment



SPRINT: 30 days

Daily SCRUM
15 min.

Scrum: organization

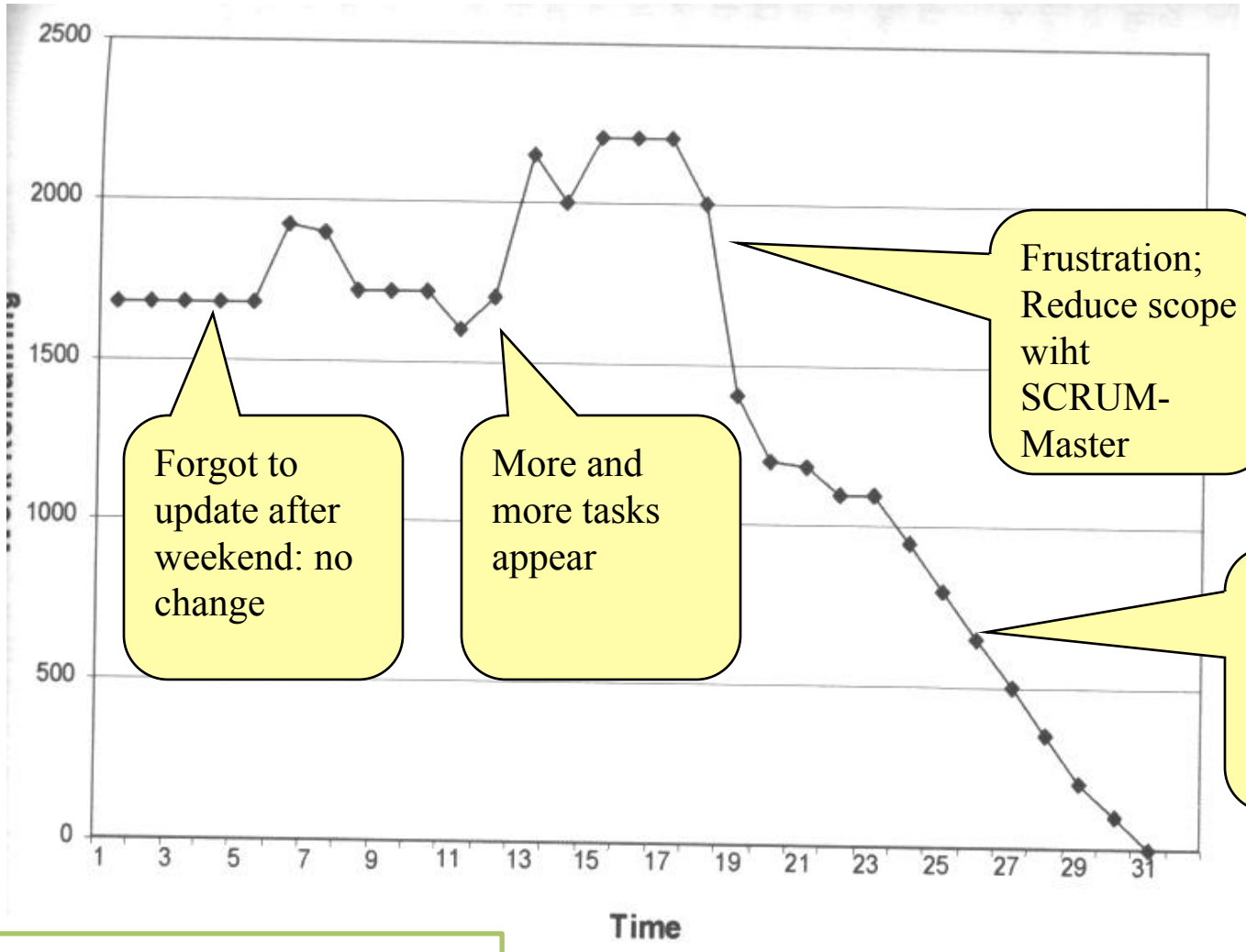


SCRUM vs. XP

- XP is often hard to introduce
- SCRUM is easy to introduce (according to Schwaber)
- Best-practice: Combine!
  - SCRUM organizational shell: *Day-to-day management*
  - XP method of implementation
  - Shared values with XP
    - Quickly generate executable code
    - Facilitate communication

Assess Sprint Progress

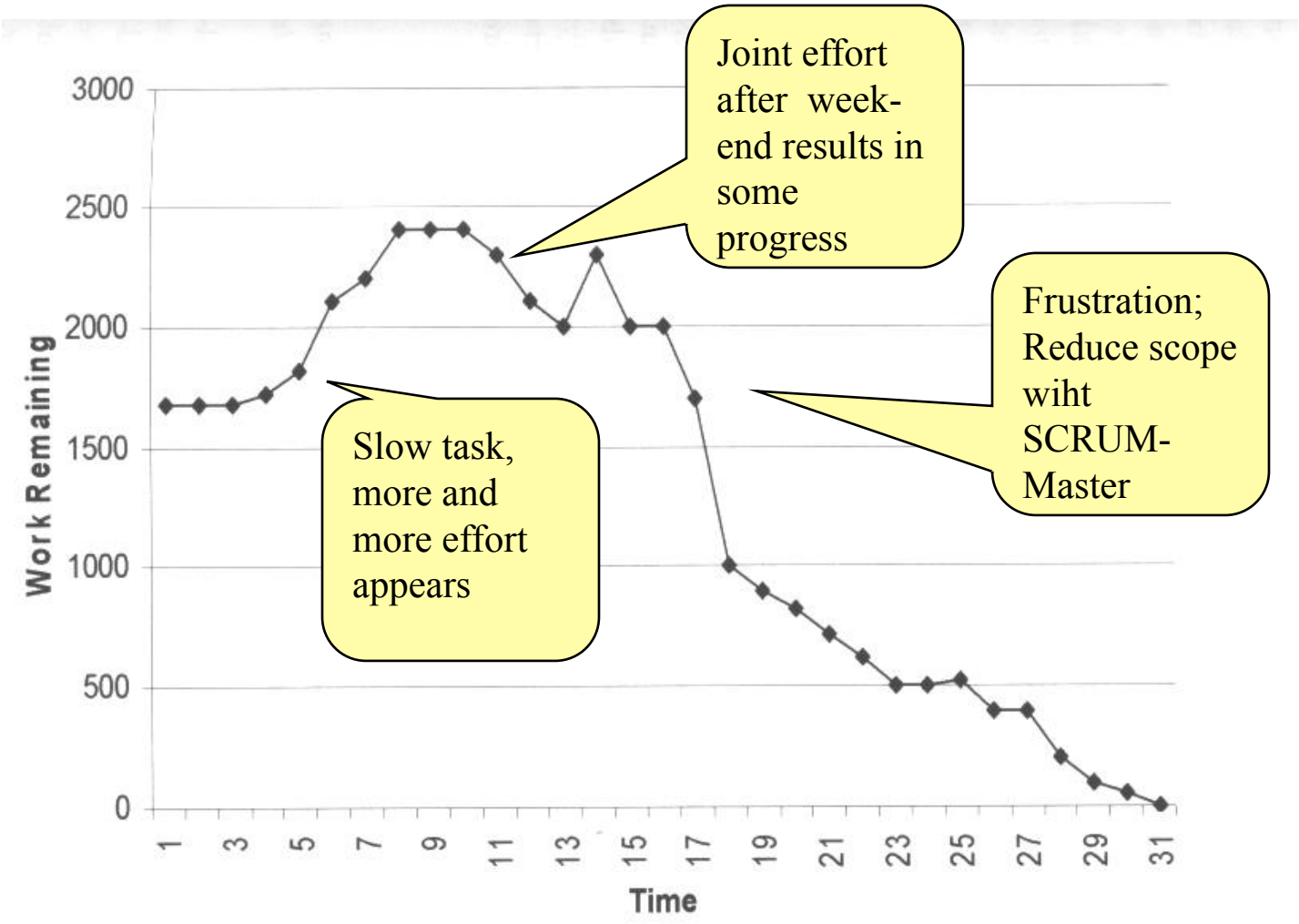
A possible trajectory



c.f. Schwaber, Ken; Beedle, Mike (2002):
Agile Software Development with Scrum. Prentice Hall.

Assess Sprint Progress

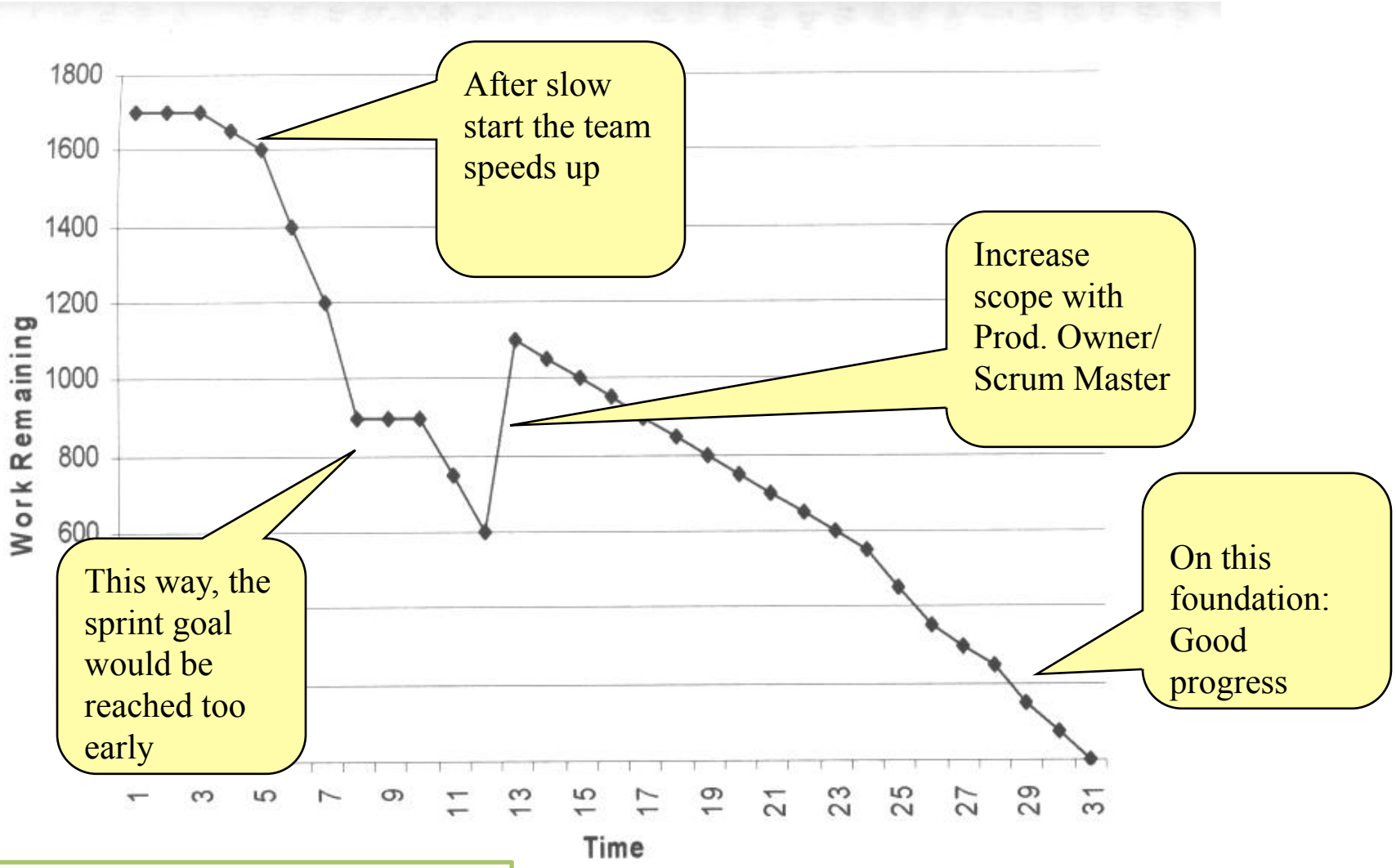
a typical trajectory for a new SCRUM-Team



C.f. Schwaber, Ken; Beedle, Mike (2002):
Agile Software Development with Scrum. Prentice Hall.

Assess Sprint Progress

another typical trajectory



C.f. Schwaber, Ken; Beedle, Mike (2002):
Agile Software Development with Scrum. Prentice Hall.

Why does SCRUM work?

- Integrated instability
  - Not too smoothly
- Self-organizing teams
  - Take ownership
- Multi-Learning
  - Between functions
  - Between group, organization, and individual
- Subtle controll
- Constant learning
  - Experienced developers in new teams

Risk management

- Risk: Customer unhappy
  - Show working system often
- Risk: Incomplete feature set
  - Prioritize: If something is missing, it is not important
- Risk: Bad estimation
  - Daily updates during SCRUM
- Risk: Lack of experience with Development cycle
  - Test early and execute repeatedly
- Risk: Changes in performance estimation
  - No impact on Sprint

c.f. Schwaber, Ken; Beedle, Mike (2002):
Agile Software Development with Scrum. Prentice Hall.

Summary SCRUM

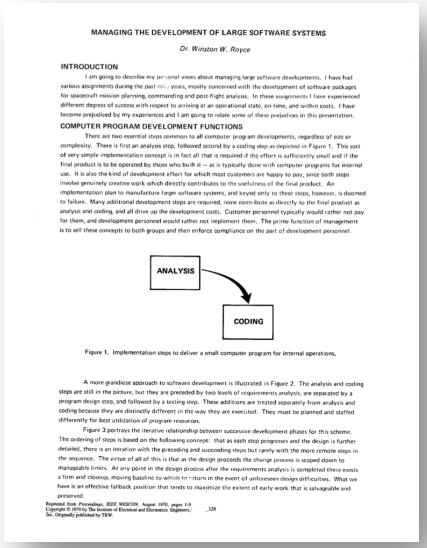
- SCRUM is a management shell
  - Around XP
  - Or other approach: Even waterfall possible
- Overlap with XP, but differences exist
  - Similar values
  - Different practices
  - Partly complement each other
- Not as much impact as XP, easier to introduce
- Strength
  - Information flows not only in one direction
  - Multiple feedback cycles stabilize system

TODO for next week

- Organize your groups
  - Choose an agile coach / scrum master / team spokesperson and make it known to Emil and me
  - Create a wiki page on https://github.com/oerich/EDA397
  - Link to your group's project on github
  - Get accounts for Pivotal Tracker for each group member

- Prepare for customer interview

Thanks! (...and optional further reading)



MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS

Dr. Winston W. Royce

INTRODUCTION

I am going to describe my personal views about managing large software developments. I have had various assignments during the past nine years, mostly concerned with the development of software packages for spacecraft mission planning, commanding and post-flight analysis. In these assignments I have experienced different degrees of success with respect to arriving at an operational state, on time, and within costs. I have become prejudiced by my experiences and I am going to relate some of these prejudices in this presentation.

COMPUTER PROGRAM DEVELOPMENT FUNCTIONS

There are two essential steps common to all computer program developments, regardless of size or complexity. There is first an analysis step, followed second by a coding step as depicted in Figure 1. This sort of very simple implementation concept is in fact all that is required if the effort is sufficiently small and if the final product is to be operated by those who built it — as is typically done with computer programs for internal use. It is also the kind of development effort for which most customers are happy to pay, since both steps involve genuinely creative work which directly contribute to the usefulness of the final product. An implementation plan to manufacture larger software systems, and keyed only to these steps, however, is doomed to failure. Many additional development steps are required, none contribute as directly to the final product as analysis and coding, and all drive up the development costs. Customer-personnel typically would rather not pay for them, and development personnel would rather not implement them. The prime function of management is to sell these concepts to both groups and then enforce compliance on the part of development personnel.

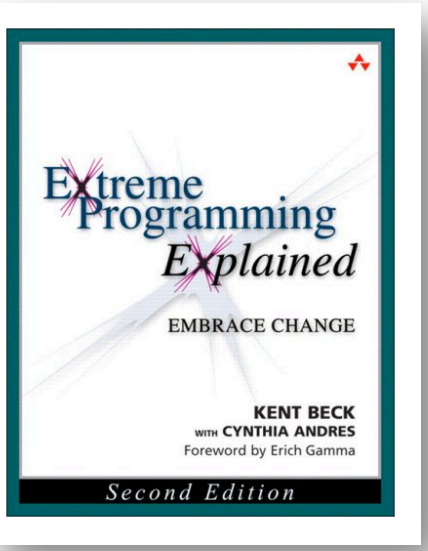
ANALYSIS → CODING

Figure 1. Implementation steps to deliver a small computer program for internal operations.

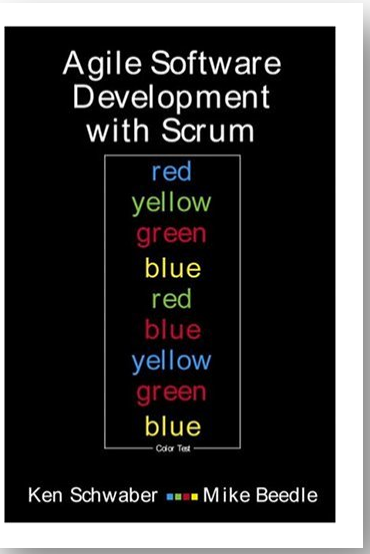
A more grandiose approach to software development is illustrated in Figure 2. The analysis and coding steps are still in the picture, but they are preceded by two levels of requirements analysis, are separated by a program design step, and followed by a testing step. These additions are treated separately from analysis and coding because they are distinctly different in the way they are executed. They must be planned and staffed differently for best utilization of program resources.

Figure 3 portrays the iterative relationship between successive development phases for this scheme. The ordering of steps is based on the following concept: that as each step progresses and the design is further detailed, there is an iteration with the preceding and succeeding steps but rarely with the more remote steps in the sequence. The virtue of all of this is that as the design proceeds the change process is scoped down to manageable limits. At any point in the design process after the requirements analysis is completed there exists a firm and cleanup, moving baseline to which to return in the event of unforeseen design difficulties. What we have is an effective fallback position that tends to maximize the extent of early work that is salvageable and preserved.

Winston W. Royce: Managing the Development of Large Software Systems. In: Proceedings of IEEE WESCON, pages 1-9, 1970



Kent Beck: Extreme Programming Explained. Addison-Wesley, 2000



Ken Schwaber and Mike Beedle: Agile Software Development with Scrum, Prentice Hall, 2002