Lecture 12:
Agile/Lean Methods

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Fall 2016
Schedule of Lectures

Week 01: Introduction to SE
Week 02: Requirements Engineering I
Week 03: Requirements Engineering II
Week 04: Analysis
Week 05: Development Infrastructure I
Week 06: Development Infrastructure II
Week 07: Architecture and Design
Week 08: Refactoring
Week 09: Verification and Validation I
Week 10: **Guest Lecture I (Testing)**
Week 11: **Guest Lecture II (Software Quality Management)**
Week 12: no lecture
Week 13: Verification and Validation II + Agile/Lean Methods
Week 14: **Guest Lecture III (Software Craftsmanship)**
Week 15: Course wrap-up, review and exam preparation
Week 16: no lecture
Structure of Lecture 12

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
Process Types

- Waterfall
- Incremental
- Iterative/Evolutionary
Heavy-Weight Waterfall Process

[Diagram of the Heavy-Weight Waterfall Process]

- Problem Recognition
- System Specification
- System Acceptance
- Completely Operational System
- Obsolescence
- Analysis
- Design
- Implementation
- Installation
- Maintenance
- Traditional Concepts of Programming Cover This Span

_IN PRACTICE PROGRAMMERS PERFORM DUTIES OVER THIS SPAN_

SCHEMATIC FOR THE TOTAL PROGRAMMING (SOFTWARE SYSTEM-BUILDING) PROCESS
Process Types

- Waterfall
- Incremental
- Iterative/Evolutionary

Challenges:
- Changing Requirements
- Fixed-price/fixed-scope/fixed-deadline projects
- Heavy-weight process models (prescriptive)
- Taylorism, trying to create many specialised roles
The Agile Manifesto

Kent Beck et al. (2001):

**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.
Structure of Lecture 11

• Light-weight processes / Evolutionary development
• Agile Processes/Methods
  • Extreme Programming (XP)
  • Scrum (intro)
• KANBAN
• Lean Processes/Methods
13 XP Practices

**Project Cycle**
- Planning Poker
- Small Releases
- Whole Team
- Customer Tests

**Development Cycle**
- Simple Design
- Pair Programming
- TDD (Unit Test)
- Refactoring

**Supporting Practices**
- Coding Standard
- Sustainable Pace (40-hour week)
- Metaphor (Common Understanding)
- Continuous Integration
- Collective Ownership
Requirements vs. User Stories

Traditional requirement – “shall” statements:

• “The system shall provide a user configurable interface for all user and system manager functions”

• “The user interface shall be configurable in the areas of:
  • Screen layout
  • Font
  • Background and text color

Corresponding “User Story”:

• “As a system user or system manager, …

• … I want be able to configure the user interface for screen layout, font, background color, and text color, …

• … So that I can use the system in the most efficient manner”

who - what - why
Planning Poker

Participants in planning poker include all of the developers on the team

• **Step 1:** Give each estimator a deck of cards

• **Step 2:** Moderator reads description of User Story to be estimated.

• **Step 3:** Product owner answers any question the estimators may have about the User Story.

• **Step 4:** Each estimator privately selects a card representing his or her estimate. Cards are not shown until each estimator has made a selection.
Planning Poker (cont’d)

• **Step 5:** When everyone has made an estimate, the cards are simultaneously turned over.

• **Step 6:** If estimates differ, the highest and lowest estimates are explained by the estimators - otherwise the estimation is completed for this User Story.

• **Step 7:** The group can discuss the story and their estimates for a few more minutes. The moderator can take any notes he/she thinks will be helpful when this story is being programmed and tested. After the discussion, each estimator re-estimates by selecting a card.

-> Go to Step 5.

Note: In many cases, the estimates will already converge by the second round. But if they have not, continue to repeat the process. The goal is for the estimators to converge on a single estimate that can be used for the story. It rarely takes more than three rounds, but continue the process as long as estimates are moving closer together.
**Simple Design**

Characterisation:
- Four characteristics of simple design, listed in priority order:
  1. The system runs all the tests.
  2. It contains no duplicate code.
  3. The code states the programmers’ intent very clearly.
  4. It contains the fewest possible number of classes and methods.

**Guidelines to help in arriving at a simple design:**
- Look for a simple – but not stupid – way to solve a problem. Pay attention to good design principles when forming a system incrementally. (→ design patterns)
- Don’t add infrastructure or other features that might be needed later. Chances are they won’t be (YAGNI: You Aren’t Going to Need It). Let the user stories force you to change the design.
- Don’t generalize a solution until it is needed in at least two places. Follow the first rule above and keep implementation simple. Let the second user pay for the generality.
- Seek out and destroy duplication and other ‘code smells’ (or: ‘design smells’). The practice of refactoring is the most powerful tool in the arsenal. It is through removing duplication that new classes, methods, and larger scale systems are born.
- Remember that it is just code. If it is getting overly complex and painful, delete it. It can always be recreated again in less time and better than the first time by leveraging what was learned the first time.
Test-Driven-Development

Developer TDD => Unit Tests
Test-Driven-Development

Developer TDD =>
Unit Tests

Acceptance TDD =>
Acceptance Tests
also called:
Behavior-driven testing (BDD)
Test-Driven Development (TDD)

- Unit Test
- Functionality-oriented
- Regression testing can be automated

```
namespace UnitTestingExamples.Tests
{
    using System;
    using NUnit.Framework;
    [TestFixture]
    public class BankAccountTests
    {
        [Test]
        public void TestDeposit()
        {
            BankAccount account = new BankAccount();
            account.Deposit( 125.0 );
            account.Deposit( 25.0 );
            Assertion.AssertEquals( 150.0, account.Balance );
        }
    }
}
```

```
namespace UnitTestingExamples.Library
{
    using System;
    public class BankAccount
    {
        private double _balance = 0.0;
        public void Deposit( double amount )
        {
            _balance += amount;
        }
        public double Balance
        {
            get { return _balance; }
        }
    }
}
```
TDD & Refactoring Integrated

User Story (new functionality)

Write a failing test.

Clean up code.

Make the test pass.

Test

Refactor

Code
Refactoring

- Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. (Invented by Martin Fowler)
- Many refactorings can be automated
- Catalogue of refactorings:
  
  http://www.refactoring.com/catalog/index.htm

- Note: It is not always clear
  - (a) how to detect refactoring opportunities and
  - (b) what refactoring(s) are most appropriate

Pair Programming

Characterisation:

• Two programmers work together at one computer
  • The driver, writes code
  • The observer (or navigator), reviews each line of code as it is typed in
• The two programmers switch roles frequently
Pair Programming

Characterisation:
• Two programmers work together at one computer
  • The **driver**, writes code
  • The **observer** (or navigator), reviews each line of code as it is typed in
• The two programmers switch roles frequently

Benefits:
• Studies found that programmers working in pairs produce
  • shorter programs,
  • with better designs
  • and fewer bugs
  faster

Challenges:
• **Total effort (person-hours) increases**
  • Management needs to balance faster completion of the work and reduced testing and debugging time against the higher cost of coding
  • Greatest benefit on tasks that the programmers do not fully understand before they begin: that is, challenging tasks that call for creativity and skill
    • On simple tasks, which the pair already fully understands, pairing results in a net drop of productivity.
• Productivity might drop when novice-novice pairing is used without coaching.
Structure of Lecture 11

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
The Term “Scrum”

Originates from Rugby
Meaning “crowded”
Complex move that requires team work
What is Scrum?

• Agile Management Framework for SW development projects

• With a few clear rules:
  • Roles: Product Owner, Team, Scrum Master
  • Product Backlog, Sprint Backlog, few compact reports
  • Short work cycles (→ ”Sprints”) for incremental development

• Based on the Agile Manifesto of Kent Beck at al.
  • Human-centred
  • Technology and tools have secondary role
  • Close cooperation with customer
  • Empirical learning process

Scrum does not define a development methodology, QA strategy, or risk management approach, but asks the team to take care of these issues appropriately.
Scrum Elements – Process, Artifacts, Roles

- Preparation
  - Business case & funding
  - Vision
  - Initial product backlog
  - Initial release plan
  - Stakeholder buy-in
  - Assemble team
  - Logistics

- Process
  - Sprint planning meeting
  - Update product backlog
  - Sprint retrospective
  - Sprint review
  - Daily scrum
  - Daily work
  - Product increment

- Release n

- Scrum artefacts
  - Product backlog
  - Impediment list
  - Sprint backlog delta report
  - Sprint backlog burndown
  - Product backlog burndown

- Scrum roles
  - Product owner
  - Scrum master
  - Users
  - Team members
  - Stakeholders

http://www.scrumforteamsystem.com/processguidance/v1/Scrum/Scrum.html
Scrum Process – Simplified Overview

- Product Owner
- Product Backlog
- Sprint Backlog
- Sprint
- Team
- ScrumMaster
- Shippable Product Increment
Scrum: Backlogs

Product Backlog
• Collection of requirements (user stories) for the product – at project start
  • a few, little detailed user stories; collection evolves over time and requirements will be refined over time
• Managed by the Product Owner

Sprint Backlog
• Collection of requirements (user stories) that are selected for implementation during next sprint
• Managed by the Team
Scrum: Sprint

Sprint

- Period (2-4 weeks) in which a shippable product increment (executable, tested, and documented) is created by the Team
- Time-boxed
  - i.e., ends exactly at the scheduled time
- At end of Sprint: Product Owner accepts/rejects the final results (i.e., the software)
  - Partially completed or incorrect results
    - will not be shipped (no compromise on quality) and
    - go back to the Product Backlog for inclusion in the next Sprint (Backlog)
Scrum: 3 Roles

**Product Owner**
- Decides which requirements are implemented for a product version
- Decides about when product increments will be shipped

**Team**
- Implements requirements
- Decides how many requirements are implemented in a Sprint
- Organizes its activities (→ tasks) independently

**Scrum Master**
- Takes care of the proper implementation of Scrum
- Supports the team in process-related issues
Sprint Meetings

Sprint Flow

1 day for a 4 week Sprint
15 min daily

~1 hour

Sprint Planning Meeting
Sprint Review Meeting
Sprint Retrospective
Sprint Planning Meeting
Sprint Burndown Chart – Example
Scalability of Scrum

- Typical individual team is 7 ± 2 people
  - Scalability comes from teams of teams
- Factors in scaling
  - Type of application
  - Team size
  - Team dispersion
  - Project duration
- Scrum has been used on multiple 500+ person projects (e.g., SAP)
Structure of Lecture 11

- Light-weight processes / Evolutionary development
- Agile Processes/Methods
  - Extreme Programming (XP)
  - Scrum (intro)
- KANBAN
- Lean Processes/Methods
Kanban (Jap.): literally ‘signboard’ or ‘billboard’

**SCRUM** vs. **KANBAN**

- Time-Boxing vs. Task-Boxing
- Velocity vs. Lead-Time
<table>
<thead>
<tr>
<th>Selected (6)</th>
<th>Analysis (1)</th>
<th>Development</th>
<th>Staging (1)</th>
<th>Verification</th>
<th>Scheduled (10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>Story (3)</td>
<td>Wait (*)</td>
<td>Implement (4)</td>
<td>Ready (4)</td>
<td>Testing (4)</td>
</tr>
<tr>
<td>Ready</td>
<td>Ready (4)</td>
<td>In progress (4)</td>
<td>Ready (4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### User Story
- As a user I can add an image in news wizard
- As a user I want to change my password
- User des Database
- As a user I want to add a new article
- As a publisher I can add a new article
- Article type to wizard
- Database changes
- Article icon
- Privilege check
- Fix server
- Fix JavaScript
- Partial build
- Link to my page
- Business logic
- As a developer I want to fix the feature
- As an administrator I want to verify the changes
- Business logic
- As a question
- Task X
- Task Y
- As a user
- As a user
- As a user
- As a user
- Task Z

### Suggestions (2)
- Production problem
- GetFeatureInfo does not work
More prescriptive

RUP (120+)

XP (13)

Scrum (9)

Kanban (3)

Do Whatever (0)

More adaptive

Visualize Your Workflow
Limit Your WIP
Use Lead-Time as default metric

RUP has over 30 roles, over 20 activities, and over 70 artifacts.
Structure of Lecture 11

• Light-weight processes / Evolutionary development
• Agile Processes/Methods
  • Extreme Programming (XP)
  • Scrum (intro)
• KANBAN
• Lean Processes/Methods
Origins of Lean Software Development

• Originates from Toyota Production System (TPS)
  • Also called Just-In-Time system

• Post WWII Japanese automobile industry could not compete with U.S. mass production systems

• Inspiration for TPS found in the 1950’s from U.S. supermarkets
  • Customers could get what they wanted, when they wanted it and shelves were refilled when items were about to run out.

• The concepts transferred to the domain of software engineering by Mary and Tom Poppendieck (2003, 2007).
Main Goals of LEAN

1. All processes shall give value
   • Remove everything that does not create value

2. Ensure good flow in the processes to avoid bottlenecks and queues (-> work not piling up & waiting)

3. All activity shall be based on need (-> Pull)
   • If there is no demand for a product or service, the related task is unnecessary

4. Become a learning organization with focus on continuous stepwise improvement
   • Kaizen (= small change for the better)
Focus on reducing the activities that do not create value

The approach to continuous improvement

Focus on removing/reducing the activities that do not create value for our customers

Traditional approach

Focus on the efficiency of the activities that create value for the customers
Seven Wastes of Software Development

- **Handoffs.** Passing the information/work to someone else, getting information/work from someone else.
- **Partially done work.** Something that is not done. E.g. untested code, undocumented or not maintained code.
- **Task switching.** How many other tasks people need to do. E.g. the amount of projects done simultaneously.
- **Delays.** Waiting for something.
- **Extra features.** Something that is not really needed.
- **Defects.** Something that does not meet the targets, or is not what it is supposed to be. E.g. software bugs, incorrectly implemented business requirements.
- **Relearning (waste of knowledge).** E.g. forgetting decisions, re-trying solutions already tried, the inability to utilize the knowledge of other people.
Next Lecture

• Date/Time:
  • Friday, 02-Dec, 10:15-12:00

• Topic:
  • Software Craftsmanship, the Codeborne Way by Anton Keks, Codeborne

• For you to do:
  • Finish and submit Lab Task 6 on time!