SOFTWARE ENGINEERING

Week 3

Introduction – Agile Software Processes

Asst. Prof. Dr. A. Cüneyd TANTUĞ Asst. Prof. Dr. Tolga OVATMAN
Istanbul Technical University
Computer Engineering Department

Agenda

1. Agility
2. Common Agile Practices
3. Extreme Programming
4. Scrum
5. Kanban - Scrumban

1. Agility
2. Common Agile Practices
3. Extreme Programming
4. Scrum
5. Kanban

Agility

3.1

Rapid Software Development

- Rapid development and delivery is now often the most important requirement for software systems
  - Specification, design and implementation are inter-leaved
  - System is developed as a series of versions with stakeholders involved in version evaluation
  - User interfaces are often developed using an IDE and graphical toolset.

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 et al.

www.agilemanifesto.org

Principles of Agile Methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

1. Agility
2. Common Agile Practices
3. Extreme Programming
4. Scrum
5. Kanban

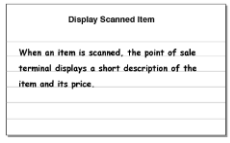
Common Agile Practices

3.2

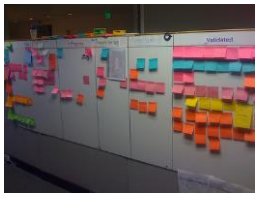
User Stories

- In most of the agile methods, a customer or user is part of the team and is responsible for making decisions on requirements.
- User requirements are expressed as scenarios or user stories.
- These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

Story Boards


Display Scanned Item

When an item is scanned, the point of sale terminal displays a short description of the item and its price.



Story Points

- Story point is an arbitrary measure used to measure the effort required to implement a story.
- It is a number that tells the team how hard the story is. Hard could be related to **complexity**, **unknowns** and **effort** but **not time**

1, 2, 4, 8, 16, 32, 64

XS, S, M, L, XL, XXL

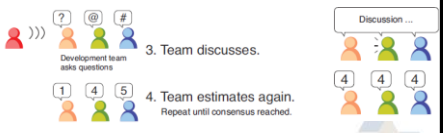
1, 2, 3, 5, 8, 13, 21, 34, 45

1, 5, 10, 25, 50, 100

- For every team, story size could mean different things depending on what baseline they chose

Planning Poker

- Planning poker is a game where the development team estimates stories individually first and then compares the results collectively together after
- If everyone's estimate is more or less the same, the estimate is kept. If there are differences, however, the team discusses them and estimates again until consensus is reached.



Refactoring

- Refactoring is the process of changing the design of your code without changing its behavior—what it does stays the same, but how it does it changes.
- You can think of it like improving a function-method-class without changing its interface.
- However, some changes requires architecture refactoring and this is much more expensive.
- Examples of refactoring:
  - Re-organization of a class hierarchy to remove duplicate code.
  - Tidying up and renaming attributes and methods to make them easier to understand.
  - The replacement of inline code with calls to methods that have been included in a program library.

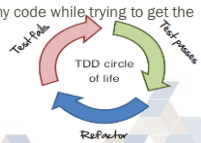
Pair Programming Environment

Advantages of Pair Programming

- It supports the idea of collective ownership and responsibility for the system.
  - Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- It acts as an informal review process because each line of code is looked at by at least two people.
- It helps support refactoring, which is a process of software improvement.
  - Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

Test Driven Development

- Test-driven development (TDD) is a software development technique that uses really short development cycles to incrementally design your software.
  - Red: Before you write any new code for the system, you first write a failing unit test, showing the intent of what you would like the new code to do. Here you are thinking critically about the design.
  - Green: Then you do whatever it takes to make the test pass. If you see the full implementation, add the new code. If you don't, do just enough to get the test to pass.
  - Refactor: Then you go back and clean up any code while trying to get the test to pass.



Collective Ownership

- Collective code ownership spreads responsibility for maintaining the code to all the programmers. Collective code ownership is exactly what it sounds like: everyone shares responsibility for the quality of the code.
- No single person claims ownership over any part of the system, and anyone can make any necessary changes anywhere.
- Collective code ownership requires letting go of a little bit of ego. Rather than taking pride in your code, take pride in your team's code.

  Always leave the code a little better than you found it.

Collective Ownership

- How can you take ownership of code that you don't understand?
  - take advantage of pair programming
  - Rely on the unit tests for further documentation and as your safety net
  - As you work, look for opportunities to refactor the code.
- Collective code ownership shares knowledge and improves skills, but it won't make everyone an expert at everything. Take advantage of all skills and specialties.
- Rather than turning your junior programmers loose on the code, make sure they pair with experienced members of the team.
- Collective ownership increases the possibility of merge conflicts, and so it also requires continuous integration. Continuous integration decreases the chances of merge conflicts.

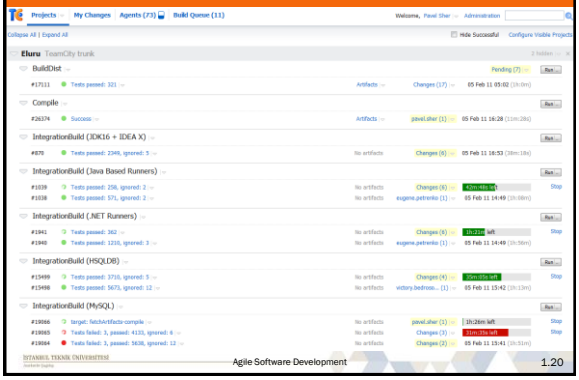
Continous Integration

- The ultimate goal of continuous integration is to be able to deploy all but the last few hours of work at any time.
- Most software development efforts have a hidden delay between when the team says "we're done" and when the software is actually ready to ship.
- The point is to be technologically ready to release even if you're not functionally ready to release.
  - Integrate your code every few hours.
  - Keep your build, tests, and other release infrastructure up-to-date.
- To guarantee an always-working build
  - make sure what works on your computer will work on anybody's computer.
  - nobody gets code that hasn't been proven to build successfully.

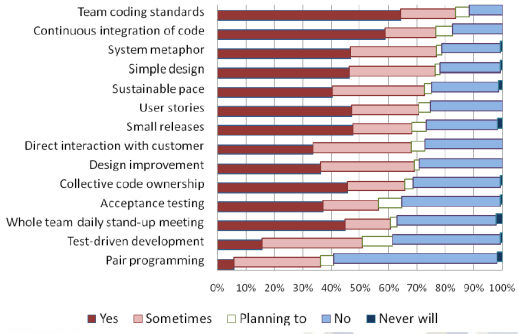
Continous Integration

- There's a lively community of open-source continuous integration servers
  - CruiseControl
  - Jenkins
  - TeamCity

Continous Integration



Overall Usage



Overall Usage

Table 9. Usage of specific agile practices

Practices	n	Mean	Median
Prioritized work list	204	4,2	4
Iteration/sprint planning	203	4,1	4
Daily stand-up meetings	209	3,7	4
Unit testing	199	3,7	4
Release planning	196	3,9	4
Active customer participation	196	3,5	4
Self-organizing teams	194	3,5	4
Frequent and incremental delivery of working software	189	4,1	4
Automated builds	185	3,5	4
Continuous integration	182	3,8	4
Test-driven development (TDD)	179	2,7	3
Retrospectives	177	3,6	4
Burn-down charts	174	3,2	3
Pair programming	174	2,4	2
Refactoring	163	3,4	3
Collective code ownership	159	3,3	3
Other	9	1,8	1

1. Agility
2. Common Agile Practices
3. Extreme Programming
4. Scrum
5. Kanban

Extreme Programming

3.3

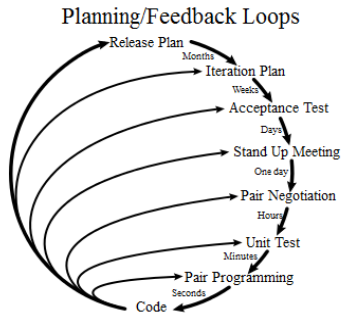
Extreme Programming

- The most widely used agile process, originally proposed by Kent Beck
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
  - New versions may be built several times per day;
  - Increments are delivered to customers every 2 weeks;
  - All tests must be run for every build and the build is only accepted if tests run successfully.

XP Principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- People not process through pair programming, collective ownership and a process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.

XP loops



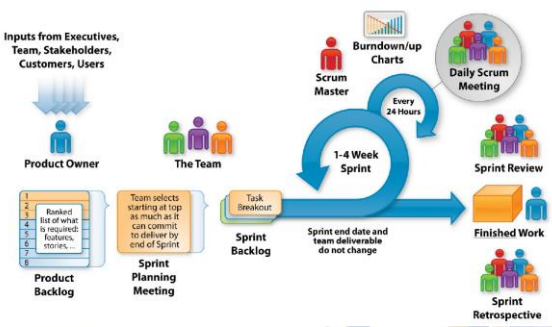
XP Phases

- XP Planning
  - Begins with the creation of "user stories"
  - Agile team assesses each story and assigns a cost
  - Stories are grouped to for a deliverable increment
  - A commitment is made on delivery date
  - After the first increment "project velocity" is used to help define subsequent delivery dates for other increments
- XP Design
  - Follows the KIS principle
  - Encourage the use of CRC cards
  - For difficult design problems, suggests the creation of "spike solutions"—a design prototype
  - Encourages "refactoring"—an iterative refinement of the internal program design
- XP Coding
  - Recommends the construction of a unit test for a store *before* coding commences
  - Encourages "pair programming"
- XP Testing
  - All unit tests are executed daily
  - "Acceptance tests" are defined by the customer and executed to assess customer visible functionality

Scrum

3.4

Scrum Process



Scrum Process

- Scrum Events
  - Sprint Planning Meeting
  - Daily Scrum
  - Sprint Review
  - Sprint Retrospective
- The Scrum Team
  - The Product Owner
  - The Development Team
  - The Scrum Master
- Scrum Artifacts
  - Product Backlog
  - Sprint Backlog
  - Increment

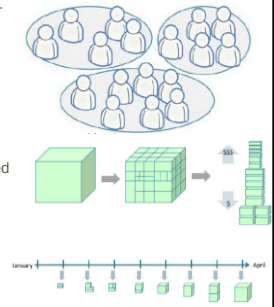
Scrum Master

Scrum Master performs following duties

- Service to the Product Owner
  - Finding techniques for effective Product Backlog management;
  - Clearly communicating vision, goals, and Product Backlog items to the Development Team;
  - Understanding long-term product planning in an empirical environment;
- Service to the Development Team
  - Coaching the Development Team in self-organization and cross-functionality
  - Removing impediments to the Development Team's progress;
  - Facilitating Scrum events as requested or needed
- Service to the Organization
  - Leading and coaching the organization in its Scrum adoption and implementations
  - Causing change that increases the productivity of the Scrum Team
  - Working with other Scrum Masters

Scrum in a Nutshell

- Split your organization into small, cross-functional, self-organizing teams.
- Split your work into a list of small, concrete deliverables. Sort the list by priority and estimate the relative effort of each item.
- Split time into short fixed-length iterations (usually 1 – 4 weeks), with potentially shippable code demonstrated after each iteration.
- Optimize the release plan and update priorities in collaboration with the customer, based on insights gained by inspecting the release after each iteration.
- Optimize the process by having a retrospective after each iteration.



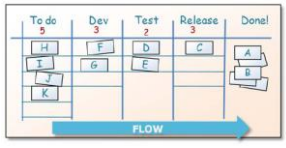
1. Agility
2. Common Agile Practices
3. Extreme Programming
4. Scrum
5. Kanban

Kanban

3.4

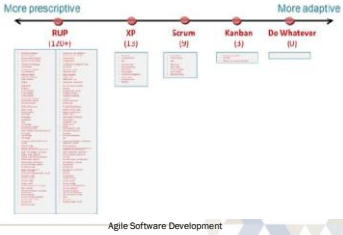
Kanban in a Nutshell

- Visualize the workflow
- Limit Work In Progress (WIP) – assign explicit limits to how many items may be in progress at each workflow state.
- Measure the lead time (average time to complete one item, sometimes called "cycle time"), optimize the process to make lead time as small and predictable as possible.



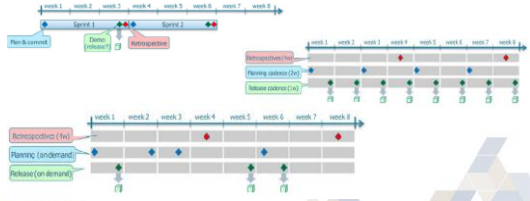
Kanban in a Nutshell

- Visualize the workflow
- Limit Work In Progress (WIP) – assign explicit limits to how many items may be in progress at each workflow state.
- Measure the lead time (average time to complete one item, sometimes called "cycle time"), optimize the process to make lead time as small and predictable as possible.



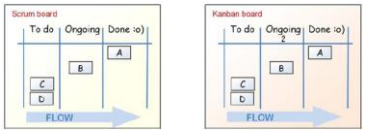
Scrum vs Kanban

- Scrum is more prescriptive
- Kanban doesn't prescribe any roles at all.
- In Kanban timeboxed iterations are not prescribed. You can go using
  - Single Scrum iterations
  - Multiple periodic cadances
  - Multiple Event-driven cadances. E.g. We trigger a release whenever there is a set of Minimum Marketable Features (MMFs) ready for release

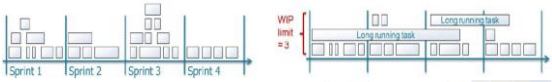


Scrum vs Kanban

- Kanban limits WIP per workflow state, Scrum limits WIP per iteration.



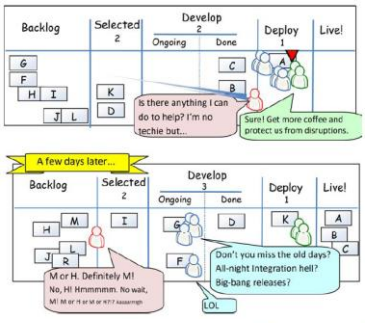
- Scrum board is reset between each sprint
- Scrum prescribes cross-functional teams, Kanban boards are cross-team
- Scrum backlog items must fit in a sprint



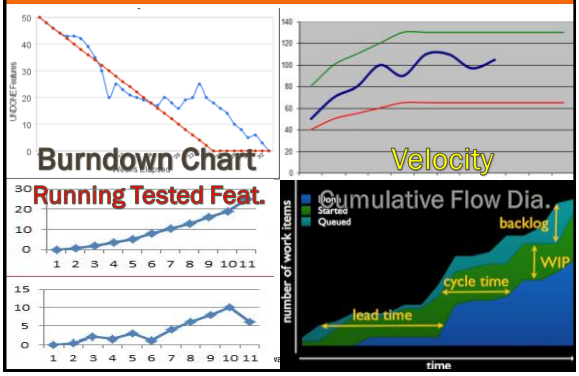
Scrum – Kanban Similarities

- Both use a pull based system
- Both allow working on multiple products simultaneously
- Both are empirical

One Day in Kanban



Scrum-Agile Metrics



Problems with agile methods

- It can be difficult to keep the interest of customers who are involved in the process.
- Team members may be unsuited to the intense involvement that characterizes agile methods.
- Prioritizing changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.

Agile Methods Applicability

- Product development where a software company is developing a small or medium-sized product for sale.
- Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
- Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.

Agile vs Plan-Driven Methods

Plan-driven development

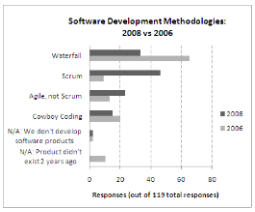
- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Not necessarily waterfall model – plan-driven, incremental development is possible
- Iteration occurs within activities.

Agile development

- Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process

Agile vs Plan-Driven Methods

Agile	Plan-driven	Formal Methods
Expert Developers	Junior Developers	Expert Developers
Frequent Changes	Rare Changes	Constrained Changes
Few Developers	Large Groups Of Developers	Modellable Requirements
Chaotic Work Environment	Tidy Work Environment	High Quality Product
Low Critical Systems	High Critical Systems	Very High Critical Systems



Wrap-up

This week we present

- The concept of agility
- Agile Practices
- Most frequently used agile methodologies
  - XP
  - Scrum
  - Kanban
- Best suggestion
  - Select practices and methodology based on organizational characteristics and experience level.

Next Week

- We will present an in-depth discussion on *Software Project Planning and Estimation!!!*