Software Development Process

Lecture 10

Lean Development

Lean Concept

- **Lean software development** has evolved from lean production process in automotive industry
- **Lean manufacturing** is a management philosophy described by Toyota Production System
- Everything that does not add any value to of the end customer is a **waste**
  - And waste is to be eliminated
- **Value** is defined as any function or process that a customer would be willing to pay for

Lean Software Development

- First described by Mary Poppendieck and Tom Poppendieck
  - Applying lean principles in software development
  - Describing **tools** or **development practices** and compare them with agile development
  - Summarize Toyota's philosophy in **7 principles**

Lean Principles

1. Eliminate waste
2. Amplify learning
3. Decide as late as possible
4. Deliver as fast as possible
5. Empower the team
6. Build integrity in
7. See the whole

Lean Waste

- In lean, there are three kinds of waste:
  - **Muda** (**waste**): Unnecessary products and works that add no value to the customer.
  - **Mura** (**unevenness**): Clogged workflow or bottleneck. Fluctuations in volumes or qualities. Can be improved by Just-In-Time systems.
  - **Muri** (**excessive work**): Poorly planned, unorganized work. Can be avoided by standardized work.

Types of Waste

- Waste or muda is everything that does not add values to the customer aka **NVA** (**Non-Value-Added**)
- **Type I Muda**: NVA tasks / items that cannot be avoided
  - Testing or inspection
  - Management activities
- **Type II Muda**: NVA that can be eliminated immediately
  - Extra processes e.g. paperworks
  - Extra unnecessary features
  - Low quality products

The Seven Muda

1. **Transport**: Having to move things that are not required out of the way
2. **Inventory**: Raw materials, work in progress or finished products that are no longer being processed
3. **Motion**: People or materials that must be moved than required to perform the processing
4. **Waiting**: Idle time needed to wait for other process to be completed

The Seven Muda (2)

5. **Overproduction**: Producing something before it is needed
6. **Over Processing**: Additional activity required due to poor design or poor tools
7. **Defects**: The effort to detect and fixing defects

Eliminate Waste

- Example of waste software development:
  - **Unnecessary codes**: Partially done codes, unnecessary functionalities, extra processes
  - **Delays**: Waiting for other activities or teams, defects or low quality products that cause delay
  - **Unclear requirements**: Poorly define / unclear requirements will lead to defects and wrong functionalities
  - **Poor testing**: This could lead to wasted work
  - **Management bureaucracy**: Extra paperwork, unnecessary documentation are all waste
  - **Poor communication**: Slow communication which can result in wasted effort or delays is a waste

Eliminate Waste (2)

- To eliminate waste, we have to recognize it first
- There are many methods to do so but we will keep it simple here:
  - Find the tasks or artifacts that when skipped, we can still complete our jobs
  - Ask yourself, why are we doing this? Does it add any value to us and/or the customer?
- Keep looking for waste

Amplify Learning

- Software development is continuous learning
  - The tasks that are assigned to a dev team could be new and unknown
  - Thus, the team might need to explore ways to resolve the tasks: **learning**
- The best approach to improve the team is to look for ways to enhance or amplify learning

Amplify Learning (2)

- **Show and tell**:
  - One can learn what the users really want by gathering feedback, by showing the customers prototypes, designs, mock-ups, or demos,
  - Various ideas can also be tried / experimented by developing a prototype and showing to the customers and getting their inputs
  - Running tests as soon as the code is done also leads to early defects detection
- **Short iterations**:
  - This approach leads to quick feedback and the team will learn more about any problems earlier
  - The customers also learn their actual needs earlier too

Amplify Learning (3)

- XP practices can also be applied here
  - Pair programming
  - Code reviews
- Other documentation
  - Manual or model
  - Commented codes
  - Training
- Retrospective sessions
  - What have we done/achieved?
  - What went well? / What worked?
  - What went wrong? / What can we improve?

Decide as Late as Possible

- Software development is always associated with uncertainty
- Therefore, late decision making would lead to better results
  - That is, all available options are kept open until we really need to pick one or when all facts are available
- The more complex the system is, the more flexible it should be
  - This would allow us to delay important decisions to the very last minute

Decide as Late as Possible (2)

- **Iterative approach**:
  - Allows the system to be flexible and adaptive.
  - The changes would be much more costly if we do it after the entire system is released and operational
- **Set-based design**:
  - If the new function is needed, we might want to try ideas all at once.
  - We could design the system or build prototypes based on different approaches.
  - Select the one that is the most suitable approach or mix and match with know-hows we learned from several approaches

Deliver as Fast as Possible

- Now, software development is not about the size, it is about speed
  - Only the fastest survives

- The sooner the products can be launched, the sooner the feedbacks can be gathered. Those would be incorporated into the next iterations

Deliver as Fast as Possible (2)

- Speed can also help delaying customer's decision. He can focus only what he wants now and does not have to care about the future (i.e. other features that are not needed yet).

- Another plus when delivering fast:
  There would be less chance or requirement changing

Deliver as Fast as Possible (3)

- Short iterations can indeed contribute to this principle
- **Just in Time** approach can also be applied
  - The customer could arrange tasks using user stories
  - The team can divide the tasks estimate efforts and allocate each (sub)tasks into iterations
  - Here, the team is changed into a self-pulling system: Tasks are being pulled into workflows (**kanban** style)

Key to Deliver Fast

- The dev team must be committed
  - Everyone in the team really wants to get the work done asap
  - When the team is waiting for someone or some work to be done, others in the team are willing to step in and help
  - “That’s not my job” is not the right mindset
- Build simple solution. Do not over-engineer
- Eliminating waste (See Principle #1)
- Have the right people (See Principle #5)

Empower the Team

- Traditionally, managers are telling the workers what to do. They are decision makers
- Lean encourages flat team structure where individuals' skills are embraced

  "find good people and let them do their own job"
- Managers' tasks are encourage progress, catching errors, removing obstacles but not micro-management

Empower the Team

- **Work Out Technique**: Managers are taught to listen to the developers. Let them explain how they do their jobs and suggest the ways to improve them.
- **Human Resources**: Human are not resources that should be told what to do. They need motivation, peer esteem and higher purpose to work for
- **Self-Organizing**: The team should be able to organize itself to achieve required solutions: Who does what and which processes and procedures to use
- **Customer-Access**: Developers need to clarify requirements, demonstrate potential solutions and get feedbacks directly from the customers

Build Integrity In

- **Integrity**: Free from flaw, defect and decay
- In other words, you want to build a sustainable system that solves the customer's problem
  - The system has no flaw: It actually solves the problems and needs.
  - It has no defect: It works as it is supposed to.
  - It does not decay over time: Even though the customer business grows or number of the users increases, the system is still working

Two Kinds of Integrity

- **Perceived integrity**: How the customer understand / perceive the system; how it is used, deployed, accessed and advertised and priced
- **Conceptual integrity**: How well the systems' components will function together with balance between flexibility, maintainability, efficiency, and responsiveness

Building-in Integrity

- **Involving the customer**: Let the PO decide on the product backlog and getting frequent feedback
- **Test-Driven Development**: Make sure that the software has no defect
- **Continuous Integration**: Decay can be reduced if your system is integrated flawlessly and defectlessly
- **Refactoring**: Evolving your code to meet current needs and standard

See the Whole

- Software systems are not only about the sum of functionalities of all subsystems, but the products of their interactions
  - You have to be careful how the subsystems are developed and put together
- By breaking down requirements or user stories into smaller work items and by standardizing developing phases, we can focus on smaller and identify defects easier. Yet, we can still maintain the sight of the big picture

See the Whole (2)

- Larger software system may involve many teams. This would require more well-defined relationships between every party.
- Such relationship would lead to smooth component integrations
- In a long development period, stronger subcontractor network is more important that optimizing short-term profit
- Lean thinking has to be well understood by all members

See the Whole (3)

- Not only seeing the whole “product,” lean also encourage seeing the whole process
  - Is your working process optimized?
  - Are there any miscommunications and misunderstandings among teams or team members?
  - Does everyone of your team own the product?

"Think big, act small, fail fast; learn rapidly"

- See the big picture
- Aim for small increments
- Deliver fast, recognize defects early
- Amplify learning