Lean software development

Lean software development is a translation of lean manufacturing principles and practices to the software development domain. Adapted from the Toyota Production System,[1] it is emerging with the support of a pro-lean subculture within the Agile community. Lean offers a solid conceptual framework, values and principles, as well as good practices, derived from experience, that support agile organizations.

Contents

Origin

The term **lean software development** originated in a book by the same name, written by Mary Poppendieck and Tom Poppendieck in 2003.[2] The book restates traditional lean principles, as well as a set of 22 *tools* and compares the tools to corresponding agile practices. The Poppendiecks' involvement in the Agile software development community, including talks at several Agile conferences [3] has resulted in such concepts being more widely accepted within the Agile community.

Lean principles

Lean development can be summarized by seven principles, very close in concept to lean manufacturing principles:[4]

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

Eliminate waste

Lean philosophy regards everything not adding value to the customer as waste (*muda*). Such waste may include:[5]

1. Partially done work
2. Extra processes
3. Extra features
4. Task switching
5. Waiting
6. Motion
7. Defects

8. Management activities

Industry research revealed these software development wastes:[6]

1. Building the wrong feature or product
2. Mismanaging the backlog
3. Rework
4. Unnecessarily complex solutions
5. Extraneous cognitive load
6. Psychological distress
7. Waiting/multitasking
8. Knowledge loss
9. Ineffective communication.

In order to eliminate waste, one should be able to recognize it. If some activity could be bypassed or the result could be achieved without it, it is waste. Partially done coding eventually abandoned during the development process is waste. Extra processes like paperwork and features not often used by customers are waste. Switching people between tasks is waste. Waiting for other activities, teams, processes is waste. Motion required to complete work is waste. Defects and lower quality are waste. Managerial overhead not producing real value is waste.

A value stream mapping technique is used to identify waste. The second step is to point out sources of waste and to eliminate them. Waste-removal should take place iteratively until even seemingly essential processes and procedures are liquidated.

Amplify learning

Software development is a continuous learning process based on iterations when writing code. Software design is a problem solving process involving the developers writing the code and what they have learned. Software value is measured in fitness for use and not in conformance to requirements.

Instead of adding more documentation or detailed planning, different ideas could be tried by writing code and building. The process of user requirements gathering could be simplified by presenting screens to the end-users and getting their input. The accumulation of defects should be prevented by running tests as soon as the code is written.

The learning process is sped up by usage of short iteration cycles – each one coupled with refactoring and integration testing. Increasing feedback via short feedback sessions with customers helps when determining the current phase of development and adjusting efforts for future improvements. During those short sessions both customer representatives and the development team learn more about the domain problem and figure out possible solutions for further development. Thus the customers better understand their needs, based on the existing result of development efforts, and the developers learn how to better satisfy those needs. Another idea in the communication and learning process with a customer is set-based development – this concentrates on communicating the constraints of the future solution and not the possible solutions, thus promoting the birth of the solution via dialogue with the customer.

Decide as late as possible

As software development is always associated with some uncertainty, better results should be achieved with a *set-based* or *options-based* approach, delaying decisions as much as possible until they can be made based on facts and not on uncertain assumptions and predictions. The more complex a system is, the more capacity for change should be built into it, thus enabling the delay of important and crucial commitments. The iterative approach promotes this principle – the ability to adapt to changes and correct mistakes, which might be very costly if discovered after the release of the system.

> With set-based development: If a new brake system is needed for a car, for example, three teams may design solutions to the same problem. Each team learns about the problem space and designs a potential solution. As a solution is deemed unreasonable, it is cut. At the end of a period, the surviving designs are compared and one is chosen, perhaps with some modifications based on learning from the others - a great example of deferring commitment until the last possible moment. Software decisions could also benefit from this practice to minimize the risk brought on by big up-front design. Additionally, there would then be multiple implementations that work correctly, yet are different (implementation-wise, internally). These could be used to implement fault-tolerant systems which check all inputs and outputs for correctness, across the multiple implementations, simultaneously.

An agile software development approach can move the building of options earlier for customers, thus delaying certain crucial decisions until customers have realized their needs better. This also allows later adaptation to changes and the prevention of costly earlier technology-bounded decisions. This does not mean that no planning should be involved – on the contrary, planning activities should be concentrated on the different options and adapting to the current situation, as well as clarifying confusing situations by establishing patterns for rapid action. Evaluating different options is effective as soon as it is realized that they are not free, but provide the needed flexibility for late decision making.

Deliver as fast as possible

In the era of rapid technology evolution, it is not the biggest that survives, but the fastest. The sooner the end product is delivered without major defects, the sooner feedback can be received, and incorporated into the next iteration. The shorter the iterations, the better the learning and communication within the team. With speed, decisions can be delayed. Speed assures the fulfilling of the customer's present needs and not what they required yesterday. This gives them the opportunity to delay making up their minds about what they really require until they gain better knowledge. Customers value rapid delivery of a quality product.

The just-in-time production ideology could be applied to software development, recognizing its specific requirements and environment. This is achieved by presenting the needed result and letting the team organize itself and divide the tasks for accomplishing the needed result for a specific iteration. At the beginning, the customer provides the needed input. This could be simply presented in small cards or stories – the developers estimate the time needed for the implementation of each card. Thus the work organization changes into self-pulling system – each morning during a stand-up meeting, each member of the team reviews what has been done yesterday, what is to be done today and tomorrow, and prompts for any inputs needed from colleagues or the customer. This requires transparency of the process, which is also beneficial for team communication.

Empower the team

There has been a traditional belief in most businesses about the decision-making in the organization – the managers tell the workers how to do their own job. In a "Work-Out technique", the roles are turned – the managers are taught how to listen to the developers, so they can explain better what actions might be taken, as well as provide suggestions for improvements. The lean approach follows the Agile Principle[7] "find good people and let them do their own job,"[8] encouraging progress, catching errors, and removing impediments, but not micro-managing.

Another mistaken belief has been the consideration of people as resources. People might be resources from the point of view of a statistical data sheet, but in software development, as well as any organizational business, people do need something more than just the list of tasks and the assurance that they will not be disturbed during the completion of the tasks. People need motivation and a higher purpose to work for – purpose within the reachable reality, with the assurance that the team might choose its own commitments. The developers should be given access to the customer; the team leader should provide support and help in difficult situations, as well as ensure that scepticism does not ruin the team's spirit.

Build integrity in

The customer needs to have an overall experience of the System. This is the so-called perceived integrity: how it is being advertised, delivered, deployed, accessed, how intuitive its use is, its price and how well it solves problems.

Conceptual integrity means that the system's separate components work well together as a whole with balance between flexibility, maintainability, efficiency, and responsiveness. This could be achieved by understanding the problem domain and solving it at the same time, not sequentially. The needed information is received in small batch pieces – not in one vast chunk - preferably by face-to-face communication and not any written documentation. The information flow should be constant in both directions – from customer to developers and back, thus avoiding the large stressful amount of information after long development in isolation.

One of the healthy ways towards integral architecture is refactoring. As more features are added to the original code base, the harder it becomes to add further improvements. Refactoring is about keeping simplicity, clarity, minimum number of features in the code. Repetitions in the code are signs of bad code designs and should be avoided. The complete and automated building process should be accompanied by a complete and automated suite of developer and customer tests, having the same versioning, synchronization and semantics as the current state of the System. At the end the integrity should be verified with thorough testing,

thus ensuring the System does what the customer expects it to. Automated tests are also considered part of the production process, and therefore if they do not add value they should be considered waste. Automated testing should not be a goal, but rather a means to an end, specifically the reduction of defects.

See the whole

Software systems nowadays are not simply the sum of their parts, but also the product of their interactions. Defects in software tend to accumulate during the development process – by decomposing the big tasks into smaller tasks, and by standardizing different stages of development, the root causes of defects should be found and eliminated. The larger the system, the more organizations that are involved in its development and the more parts are developed by different teams, the greater the importance of having well defined relationships between different vendors, in order to produce a system with smoothly interacting components. During a longer period of development, a stronger subcontractor network is far more beneficial than short-term profit optimizing, which does not enable win-win relationships.

Lean thinking has to be understood well by all members of a project, before implementing in a concrete, real-life situation. "Think big, act small, fail fast; learn rapidly"[9] – these slogans summarize the importance of understanding the field and the suitability of implementing lean principles along the whole software development process. Only when all of the lean principles are implemented together, combined with strong "common sense" with respect to the working environment, is there a basis for success in software development.

Lean software practices

Lean software development practices, or what the Poppendiecks call "tools" are restated slightly from the original equivalents in Agile software development. Examples of such practices include:

- Seeing waste
- Value stream mapping
- Set-based development
- Pull systems
- Queuing theory
- Motivation
- Measurements
- Test-driven development
- Trunk-based development

Since Agile Software Development is an umbrella term for a set of methods and practices based on the values and principles expressed in the Agile Manifesto, Lean Software Development is considered an Agile Software Development Method.[10]

See also

- Kanban
- Kanban board
- Lean integration
- Lean services
- Scrum (development)

References

1. Yasuhiro Monden (1998), *Toyota Production System, An Integrated Approach to Just-In-Time*, Third edition, Norcross, GA: Engineering & Management Press, ISBN 0-412-83930-X.
2. Mary Poppendieck; Tom Poppendieck (2003). *Lean Software Development: An Agile Toolkit* (https://books.google.com/books?id=hQk4S7asBi4C&pg=PA182). Addison-Wesley Professional. ISBN 978-0-321-15078-3.
3. Mary Poppendieck: "The role of leadership in software development" https://www.youtube.com/watch?v=ypEMdjslEOI
4. Mary Poppendieck; Tom Poppendieck (2003). *Lean Software Development: An Agile Toolkit* (https://books.google.com/books?id=hQk4S7asBi4C&pg=PA182). Addison-Wesley Professional. pp. 13–15. ISBN 978-0-321-15078-3.

5. Mary Poppendieck; Tom Poppendieck (2003). *Lean Software Development: An Agile Toolkit* (https://books.google.com/books?id=hQk4S7asBi4C&pg=PA182). Addison-Wesley Professional. pp. 19–22. ISBN 978-0-321-15078-3.
6. Sedano, Todd; Ralph, Paul; Péraire, Cécile. "Software Development Waste" (https://www.researchgate.net/publication/313360479_Software_Development_Waste). IEEE.
7. "12 Principles Behind the Agile Manifesto - Agile Alliance" (https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/). *agilealliance.org*. 4 November 2015.
8. Mark Lines; Scott W. Ambler (2012). *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise* (https://books.google.com/books?id=SLIkMhB2ew0C&pg=PA54). IBM Press. pp. 54–. ISBN 978-0-13-281013-5.
9. Mary Poppendieck; Tom Poppendieck (2003). *Lean Software Development: An Agile Toolkit* (https://books.google.com/books?id=hQk4S7asBi4C&pg=PA182). Addison-Wesley Professional. pp. 182–. ISBN 978-0-321-15078-3.
10. "What is Agile Software Development?" (https://www.agilealliance.org/agile101/). *agilealliance.org*. 29 June 2015.

- Ladas, Corey (January 2008). *Scrumban: Essays on Kanban Systems for Lean Software Development* (http://moduscooperandi.com/modus-cooperandi-press/). Modus Cooperandi Press. ISBN 0-578-00214-0.
- Ries, Eric (September 2011). *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business. ISBN 978-0307887894.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Lean_software_development&oldid=878395515"

This page was last edited on 14 January 2019, at 16:17 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.