Lean Software Development

ORIGINAL BY CURT HIBBS AND STEVE JEWETT
UPDATED BY STEFAN THORPE, CTO AT CAYLENT

CONTENTS

- SEVEN PRINCIPLES OF LEAN SOFTWARE DEVELOPMENT
- GETTING STARTED
- BEST PRACTICE 0
- BEST PRACTICE 1: DAILY STANDUP
- BEST PRACTICE 2: AUTOMATED TESTING
- BEST PRACTICE 3: CONTINUOUS INTEGRATION
- BEST PRACTICE 4: SHORT ITERATIONS
- BEST PRACTICE 5: CUSTOMER PARTICIPATION
- SUMMARY

Lean software development is a rendering of the larger lean movement to specifically optimize the IT value stream within the application and software development domain. Factors that are used to influence and improve the manufacturing process are translated to achieve the same goals of delivering value to customers through waste elimination and continuous improvement. Though software development differs from the manufacturing context in which lean was born, it draws on many of the same principles.

SEVEN PRINCIPLES OF LEAN SOFTWARE DEVELOPMENT

Lean software development embodies seven main principles, as outlined in the book, "Implementing Lean Software Development: From Concept to Cash," by Mary and Tom Poppendieck. First published in 2006, the Poppendiecks' multiple works on lean remain topical still today as more IT teams continue to implement the lean methodology year-after-year. Each of the seven principles outlined in the book contributes to lean out software development by reducing waste and optimizing the process.

1. ELIMINATE WASTE

Waste is anything that does not contribute value to the final product and interferes with the aim of delivering value to customers. This includes, but is not limited to, inefficient processes or project churn, crossing boundaries/departments, and features that won't be used. Eliminating waste is the guiding principle in lean software development.

2. BUILD QUALITY IN

Building quality into a product means preventing defects by seeking them out within your verification process rather than using post-implementation integration and testing to detect them after the fact. Not building legacy code that lacks automated unit and acceptance tests is crucial to continuous integration and nested synchronization.

3. CREATE KNOWLEDGE

While planning is useful, it is learning that is essential. Encourage developers to build and record the knowledge necessary to develop a project. This should comprehensively include all requirements, architecture, and technologies, which are seldom known or understood completely at project startup. Creating knowledge and recording it over the course of the project ensures that the final product is in line with customer expectations.

4. DEFER COMMITMENT

Reject the idea that projects should begin with a set plan for the specification. Planning is not the same as committing. Deferring commitment is positive procrastination as more information is available at the latest possible moment before an irreversible decision needs to be made.

5. DELIVER FAST

Delivering fast puts the product in front of the customer quickly so they can provide feedback, allowing companies to take a more experimental approach to product/feature development.

Fast delivery is accomplished using short iterations that produce software in small increments by focusing on a limited number of the highest priority requirements.

6. RESPECT PEOPLE

Respecting people means giving the development team's most important resource — its members — freedom to find the best way to accomplish a task, recognizing their efforts, and standing by them when those efforts are unsuccessful. Engaged team members are a company's most sustainable competitive advantage.

7. OPTIMIZE THE WHOLE

Optimizing the whole development process generates better results than optimizing local processes in isolation, which is usually done at the expense of other local processes. Brilliant products are often the result of a unique combination of technology and opportunity which is afforded by a lean software development process.

LEAN VS. AGILE

Comparing lean and agile software development reveals they share many characteristics, including the goal of quick delivery of value to the customer, but they differ in two significant ways: scope and focus. The narrow scope of Agile focuses on processes and people through its methodology of flexibility, communication, collaboration, and simplicity. Lean focuses on the bigger picture: the context in which development occurs, delivering value quickly by focusing on the elimination of waste, and improving the workflow. As it turns out, they are complementary, and real-world processes often draw from both.

LEAN VS. DEVOPS

DevOps recognizes that to optimize software development, the walls between development and operations must be broken down. Instead of development work pushing constraints downstream to operations, DevOps shares the same lean goal of helping to accelerate the value-creating processes of IT organizations by improving the speed, productivity, and quality of value delivery by optimizing workflow. DevOps integrates a lot of lean principles, focusing on improving the cultural as well as the technical collaboration between developers and operations.

GETTING STARTED

Newcomers to lean software development sometimes have trouble implementing a lean process. The lean principles don't describe an out-of-the-box solution, so one approach is to start with an agile methodology. However, a number of methodologies exist, and choosing the right one can be difficult, as one solution does not work the same for all IT teams.

ONE STEP AT A TIME

All is not lost, though. What follows is a set of interrelated practices organized in a step-by-step fashion to allow projects to implement lean software development one step at a time.

The following practices can be stand-alone, and implementing any of them will have a positive effect on productivity. Lean software development relies on prioritization, so to deliver the best results, it's advised to work through the following practices in order to see the optimum return on investment in terms of time and energy.

The following list of five practices is preceded by two prerequisites, or "zero practices," that every software project should be doing. If your project isn't beginning with these ideals, then this is the best place to start.

BEST PRACTICE 0

Version control and scripted builds are prerequisites for other practices outlined here. They are traditionally referred to as zero practices because they need to be in place before even taking the first step toward lean software development.

VERSION CONTROL

Version control (also known as revision or source control) is a shared repository for all artifacts needed to build the project from scratch, including source code, build scripts, and storage for automated tests as they are generated. Version control maintains the latest source code so developers are all working on build systems with the most up-to-date code.

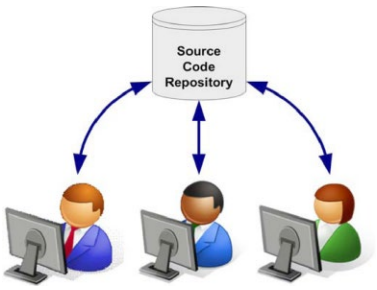

Figure 1: Centralized repository

Version control is the first prerequisite, or zero practice,

described because it is the foundation for a practical development environment, and it should be implemented before going any further.

- Select an appropriate version control system. Git is the most popular open-source tool available today and has superseded Apache Subversion as the *de facto* tool.
- Put everything needed to build the product from scratch into the version control system so that critical knowledge isn't held only by specific individuals.
- Whether centralized or distributed, team members can check code out of the version control before adding, modifying, and removing elements and before checking it back in. Updates can also be made to download any changes made by the team since the last check out.

SCRIPTED BUILDS

Scripted builds automate a build process by executing a set of commands (a script) that creates the final product from the source code stored in SCM. Scripts may be simple command files, make files, or complex builds within a tool such as Apache Maven, Apache Ant, or Gradle. The current *de facto*, but still the new kid on the block, is Docker and containers allow for one of the best scripted builds possible.

Scripted builds eliminate the potential errors of manual builds by executing the same way each time. Docker extends this capability by ensuring the container environments are identical no matter where it is deployed. They complete the basic development cycle of making changes, updating the SCM repository, and rebuilding to verify there are no errors.

- Select an appropriate build tool for your project.
- Create a script that builds the product in a uniform way from scratch, starting with source code and latest commits.

Be sure to include all the necessary key components in scripted build systems by setting all essential environment variables as well as including the data files and correct editions of the dependent libraries in version control.

A lot of scripted builds are now triggered within continuous integration tools and continuous delivery pipelines by platforms like Circle CI, Codeship, and Jenkins.

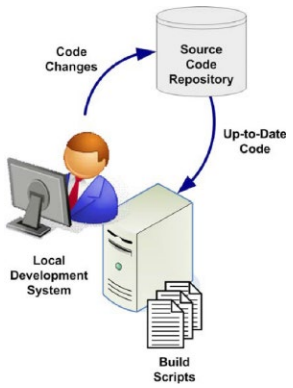

Figure 2: Zero practices

LEAN PRINCIPLES

- **Create knowledge**: Version control consolidates all team project knowledge into a single place.
- **Eliminate waste**: Manual work is eliminated by automating builds.
- **Build quality in**: Automating builds eliminates a source of errors.

BEST PRACTICE 1: DAILY STANDUP

Daily standup meetings allow each team member to communicate their work status and point out problems or issues. The meetings are short and not intended to resolve problems; rather, they serve to make all team members aware of the state of the development effort.

Borrowing from the Scrum methodology, standups are conducted by having each member of the team answer three questions:

1. What did I do yesterday?
2. What will I do today?
3. What problems do I have?

Effective daily standups result from adhering to several simple rules:

- Require all team members to attend. Anyone who cannot attend must submit their status via a proxy (another team member, email, etc.).
- Keep the meeting short, typically less than 15 minutes. Time-boxing the meeting keeps the focus on the three questions.
- Hold the meeting in the same place at the same time, every time.

- Avoid long discussions. Issues needing further discussion are addressed outside the meeting so only the required team members are impacted.

HOT TIP

The Japanese word **tsune** roughly translated means "daily habits." It refers to things such as taking a shower that are so ingrained into a daily routine that skipping them leaves one feeling that something is missing. Make the daily standup part of the team's tsune so that a day without a standup feels incomplete.

LEAN PRINCIPLES

- **Respect people**: Standups foster a team-oriented attitude; team members know what other members are doing and can get or give help as needed to move the project forward in a timely manner.
- **Create knowledge**: Sharing information engenders organizational learning by sharing group knowledge from individual knowledge.

BEST PRACTICE 2: AUTOMATED TESTING

Automated testing is essential to lean software development. There are many different types of tests that fall under the banner of automated testing. Many, if not all, have a certain place and optimal inclusion point in the build process and should be considered carefully. Automated testing includes:

- Unit testing
- Integration testing
- Smoke testing
- Acceptance testing
- Performance testing
- Load testing
- Executable specifications testing
- Story testing
- Test-first and test-driven development
- Behavior-driven development

A test framework injects predefined inputs, validates outputs against expected results, and reports the pass/fail status of the tests without the intervention of a human tester. Automated testing ensures tests are run the same way every time and are not subject to the errors and variations that can be introduced by manual testers. Such testing improves quality earlier in the process and enhances lead time.

- The basic framework for best practice automated software testing should include unit testing, smoke testing, integration testing, and acceptance testing at appropriate intervals throughout the development pipeline.
- Build a significant number of unit tests into your process early in two groups: locally run rapid sub-ten-second tests and any that take more time.
- Smoke tests should come after unit tests and they aim to spot leaks quickly in time slots of sub-15 minutes. Smoke tests are often referred to as build verification tests.
- Once early testing has been completed on the multiple components in the software, then it's time to run integration testing to verify that the varying components of the software will work as designed.
- Finally, use acceptance tests to ensure the designed software meets the customer's or business' needs. Acceptance testing also provides a forum for the client/product owner to reject different changes that have been made if they do not meet certain requirements.

HOT TIP

Developing automated tests alongside production code may be a paradigm shift for many developers. While it takes time to write tests, this time is more than recovered by eliminating rework, reducing debugging work later in the pipeline, and helping teams meet deadlines quicker. Set and adhere to new testing standards to create a culture where automated tests are the norm. This will pay off in higher quality software in the long run.

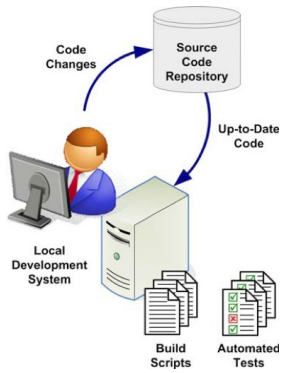

Figure 3: Automated testing

TEST EXECUTION

Each developer runs unit tests on individual code modules prior to adding them to the source code repository, ensuring all code within the repository is functional. An automated build runs both unit and integration tests to ensure changes do not introduce errors. The next practice, continuous integration, will make use of the build scripts and test suites to test the entire

system automatically each time changes are checked into the repository.

LEAN PRINCIPLES

- **Build quality in**: Automated tests are executed regularly and in a consistent manner to prevent defects and highlight any issues earlier in the process.
- **Eliminate waste**: Defects detected early are easier to correct and don't propagate or travel further down the value stream longer than necessary.
- **Create knowledge**: Tests are an effective way to document how the code functions and help team members develop their skill sets. Automated testing helps increase the depth and scope of tests to expand team knowledge.

BEST PRACTICE 3: CONTINUOUS INTEGRATION

Continuous integration (CI) is the frequent integration of small changes during the software development process. It seeks to reduce, or even eliminate, the long, drawn-out integration phase that traditionally follows a single implementation. Integrating small changes doesn't just spread the effort out over the whole cycle, it reduces the amount of integration time required because small changes are easier to ease in over a long period of time and aid debugging by isolating defects to small areas of code.

CI systems use a source code repository, scripted builds, and automated tests to retrieve source code, build software, execute tests, and report results each time a change is made.

A CI system can also check coding standards, analyze code coverage, create documentation, create deployment packages, and deploy the packages. Anything that can be automated can be included in a CI system.

- Use a dedicated build machine to host the CI system. Refer to the Continuous Integration: Servers and Tools Refcard (#087) for details on setting up a CI system.
- Check code changes into the repository a minimum of once a day (per developer); once an hour or more is even better.
- Build and test code commits in a production environment clone to mitigate errors that are only discovered on deployment due to the testing environment being significantly different to the final production environment.

HOT TIP

While the use of a dedicated computer or build machine to host the CI system may seem obvious for a large project, it provides advantages on small projects, as well:

- Dedicated machines don't compete for resources, so builds are quicker and the results get back to the developers sooner.
- Dedicated machines have a stable, well-known configuration. Builds don't fail because a new version of a library was loaded or the runtime environment was changed.

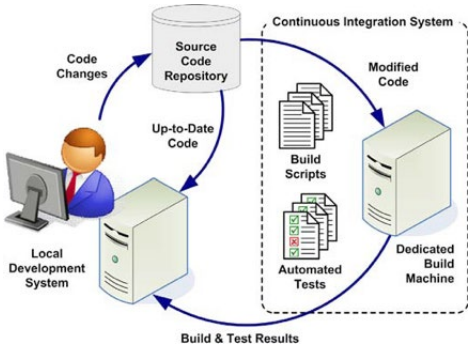

Figure 4: Continuous integration

LEAN PRINCIPLES

- **Build quality in**: Continuous integration and tests ensure code is always functional and high-quality.
- **Eliminate waste**: Frequent, small integrations are more efficient than an extended integration phase.
- **Deliver fast**: Improve lead time of feature delivery with continuous integration.

BEST PRACTICE 4: SHORT ITERATIONS

Iterations are complete development cycles resulting in the release of functional software. Traditional development methodologies often have iterations of six months to a year, but lean software development uses much shorter iterations, typically two to four weeks. Short iterations generate customer feedback quicker, and more feedback means more chances to adjust the course of development and make improvements. Deadlines are reached much more quickly and product/feature deliverables reach customers in a faster time improving customer relations with the organization.

FEEDBACK AND COURSE CORRECTIONS

Feedback from the customer is the best way to discover what's valuable to them. Each delivery of new functionality creates a new opportunity for feedback, which in turn drives course

corrections due to clarification of the customer's intent or actual changes to the requirements. Short iterations produce more feedback opportunities and allow more course corrections, so developers can hone in on what the customer wants. Ultimately, shorter iterations can have a significantly positive impact on the business' bottom line.

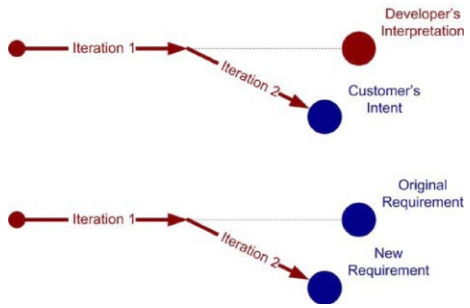

Figure 5: Course corrections

USING SHORT ITERATIONS

Several techniques aid in the implementation of a process using short iterations:

- Work requirements in priority order. High-priority requirements are typically well-defined, are the easiest to implement, and provide the most functionality in a short period of time.
- Define a non-negotiable end date for the iteration; sacrifice functionality to keep the team on schedule. End dates focus the team on delivering the required functionality. Even if some features are not completed, delivering those that are ensures customers get new functionality on a regular basis, thus improving the overall reliability and relationship between business and end user.
- Mark the end of the iteration with a demo and an official handoff to the customer. Demos foster pride in the product by allowing the team to show off its work.
- Deliver the product to the customer, whether it's a ready-for-deployment application or an interim release. Getting the product in the customer's hands for in-depth evaluation is the best way to generate feedback.

HOT TIP

Teams struggling to complete iterations successfully are often tempted to lengthen their iterations; however, longer iterations tend to hide problems. Instead, struggling teams should reduce the iteration length, which reduces the scope, focuses the team on a smaller goal, and brings impediments to the surface more quickly so they can be resolved.

LEAN PRINCIPLES

- **Eliminate waste**: Frequent, small integrations are more efficient than an extended integration phase.
- **Deliver fast**: New, functional software is delivered to the customer in closely-spaced intervals.
- **Build quality in**: Short iterations epitomize the lean software development concepts of delivering features to customers quicker in line with what they actually want and have asked for rather than what the business projects they want.

BEST PRACTICE 5: CUSTOMER PARTICIPATION

Customer participation in traditional projects typically is limited to requirements specification at the beginning of the project and acceptance testing at the end. Collaboration between customers and developers in the intervening time is limited, typically consisting of status reports and the occasional design review.

Lean software development approaches customer participation as an on-going activity spanning the entire development effort. Customers are included at all critical points of the value stream to keep product/feature development in line with the original project aims. Keeping up this line of communication with customer requirements allows developers to produce functional software from those requirements. Customers provide feedback on the software and developers act on that feedback, ensuring developers are consistently producing what the customer really wants.

INVOLVE THE CUSTOMER

The key to establishing effective customer collaboration is involving the customer in the entire development process, not just at the beginning and end. Engaging the customer, keeping them apprised of the project status, and providing a feedback path all help keep the customer involved and the development on track.

- Engage the customer by having them write and prioritize the requirements. Customers get a sense of final product/feature ownership, and they can direct the course of development profitably.
- Have the customers write the acceptance tests (or at least specify their content) and, if possible, run the tests, as well. Involvement in testing the product allows customers to specify exactly what it means to satisfy a requirement.
- Provide useful, easily accessible project status updates to keep customers in the loop, e.g. a list of the requirements in work and the status of each. Include a status on problems affecting development to avoid surprises.

- Provide access to the product at intervals so that the customer can see for themselves how it works and provide a simple, direct feedback path so that customers can input feedback easily.

COLLABORATION

Collaborating directly with the customer is necessary for developers to refine the requirements of the project and understand exactly what the customer wants.

- Designate a customer representative. The representative writes and/or collects requirements and prioritizes them accordingly. The representative clarifies any requirements for developers and provide a direct line of communication with end users.
- Schedule face-to-face time with the customer. At the very least, include a demo at the end of each iteration.

HOT TIP

Actual customers make the best customer representatives, but when customer representatives are not available, a customer proxy can fill the role. A customer proxy should be from the development team's organization and must have a good understanding of the customer's needs and the business environment.

LEAN PRINCIPLES

- **Create knowledge**: Through collaboration, requirements are discovered iteratively and refined over time.
- **Build quality in**: The end project outcome will be directly in line with customer requirements.
- **Defer commitment**: Involving customers throughout the process eliminates the need to make decisions up front.

SUMMARY

Most discussions and literature lean software development don't define specific easy-to-follow practices for implementing the process, and the large number of agile methodologies to choose from can leave newcomers confused and uncertain where to start. The specific practices outlined here provide a baseline step-by-step approach to implementing a lean software development process. Adopt one, several, or all the best practices outlined here and take your first step to streamlining your technology value stream.

REFERENCES

1. Implementing Lean Software Development: From Concept to Cash, Poppendieck/Poppendieck, Addison-Wesley Professional, 2006
2. Moving Toward Stillness, Lowry, Tuttle Publishing, 2000.
3. Emergent Design: The Evolutionary Nature of Professional Software Development, Bain, Addison-Wesley Professional, 2008

Some of the concepts and material in this Refcard were adapted from The Art of Lean Software Development, Hibbs/Jewett/Sullivan, O'Reilly Media, 2009.

Written by Stefan Thorpe

Stefan is an IT professional with 20+ years management and hands-on experience providing technical solutions to support strategic business objectives. Stefan fearlessly leads engineering teams around the globe and is responsible for platform growth and underlying technology stacks for a variety of company types. His focus and passion are on DevOps, automation, highly scalable systems, and back-end architecture/infrastructure.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300