Measuring Continuous Integration Capability

Bas Vodde
Odd-e

Continuous integration (CI) is an important Lean software development practice. Measuring the capability of your CI environment provides a road map for improvement and an aid for sharing practices between projects. The CI grid, introduced in this article, is a simple, question-based metric for checking the current CI capability of a project.

For the last few years, Lean thinking has been applied to software development, introducing Lean software development [1-3]. The roots of Lean thinking are in the Toyota Production System (TPS) [4]. The TPS has two pillars: autonomation (Jidoka) and just-in-time. Autonomation is also described as *automation with a human touch* and is based on the idea that machines automatically stop when a mistake is detected so that it can be fixed immediately and no material, effort, or electricity is wasted. A CI system brings autonomation to Lean software development.

Software integration is often a problematic area in product development. CI is a common technique used to overcome these traditional problems. CI means that developers integrate their software as frequently as possible (at least daily), in small steps (small batches), to prevent sudden surprises. Increasing the integration frequency requires making it easier to integrate, and often means reducing processes such as formal inspections and approvals. (Peer reviews and personal code reviews are certainly still valuable.) Different mechanisms need to be in place to ensure the quality of the integrations. This is where a CI system provides the support – the safety net – that enables CI. A CI system always compiles the software and runs all the tests [5]. When one step fails, the system stops like an autonomated system and will inform the person who likely broke it. Such a capability is essential in modern Lean software development. Important questions should include: What is the capability to continuously integrate in the project? How about other projects in the company?

This article introduces a grid for making the CI capability visible. This can be used for planning improvements and sharing practices.

In 2005, the grid was introduced in Nokia Networks, making telecommunications equipment, with the goal of measuring the current CI capabilities in the teams moving to Agile development. In the two years that we have used the grid, it has provided a target and visibility of improvement in the area of CI.

History

Daily building was made popular by Microsoft in the '90s and cited as a best practice in the book "Rapid Development" [6-8]. Extreme Programming, introduced in the late '90s, took daily building to the extreme and introduced the concept of CI – check-in in small steps, then compile and test everything during each check-in [9, 10]. The introduction of tools such as CruiseControl [11] makes setting up a CI environment easier and is making CI more popular.

"Software integration is often a problematic area in product development. CI is a common technique used to overcome these traditional problems. CI means that developers integrate their software as frequently as possible ... to prevent sudden surprises."

CI Grid

Background

The ultimate goal of CI is to always have a shippable, working product. Some features might not be implemented completely, but they will not break the product.

In the CI grid, we assume two levels of automated integration and testing. The first level is CI. A build in CI is triggered by a very short time-based trigger (e.g., five minutes) or by a check-in in the Software Configuration Management (SCM) system. The system is then compiled and tested. Due to the time it takes to execute all tests in a large project, it is not useful (or possible) to run all the tests. In the first level – CI – the focus is on providing quick feedback. The second level is daily builds. Daily builds are executed nightly. A daily build has a slower feedback cycle with the result being ready before the morning. Thus, in daily builds, the automated test set can be much larger and ideally contains all automated tests. The shorter CI cycle prevents the daily build from breaking frequently.

Especially in large product development, the CI and daily build can happen on both whole-product level and on subsystem level. However, only having CI and daily build on the subsystem level causes integration problems and does not create the ability to have a shippable, working version of the product every day. The feedback time for CI is essential and therefore it might be needed to have the CI on subsystem level. When having CI on subsystem level, the daily build can stay on product level and can catch subsystem integration problems early.

The described environment is definitively not the only possible way of implementing CI. Two levels might not be the best solution for very large projects and is too much for smaller projects. However, this environment offers a good starting point for most projects. If tests can be run easily all the time, then one level might be enough. If there is trouble running all tests in the daily build, then focus should be placed on speeding up the build and tests. Projects rarely need more than two levels.

CI Grid Overview

The CI grid is a tool for making CI capability visible in either an organiza-

tion or in a single project. The benefits of making this visible are:

- Give guidance for projects to improve their capability.
- Share practices between different projects. Projects can learn from each other.
- Give improvements a higher priority. (What gets measured gets done)

One warning upfront: The grid only measures the environment and capability to use CI. CI is a practice – a habit – of the development team. The grid does not measure if developers are integrating their code frequently.

The grid is a matrix with questions and metrics. Questions are answered and a color is filled into the table. There are four different possible colors:

- Red (white): Not started.
- Yellow (grey): We are working on this.
- Green (black): Yes, we have this.
- Blue (X): We have no interest in this.

Colors, not numbers, are used for getting a quick overview. (For this article, I'll use grayscale since the print is black and white; in real use, I recommend color.)

The metrics are not intended to be very precise, they can be estimated. The questions are categorized into three groups. Each group contains questions about how well a project is going in:

- Daily build.
- CI.
- Test-driven development (TDD) [12].

Daily Build Questions

The questions in the daily build section are:

1. Compilation

a. Is the whole product compiled (and linked) every day at a fixed time automatically?

If the product consists of subsystems or sub-components then the *whole product* here would mean all of these. *Automatically* means that no manual intervention is needed to start the daily build and that the build itself does not require human attention either, regardless of whether the build succeeds or fails.

2. Sanity check

a. Are essential tests run to ensure the stability of the build?

Essential tests ensure that the main functionality of the build is working. Having just the essential test requires less amount of test automation than the other questions and, thus, can be done fairly easy for projects which do not have much test automation.

3. Unit tests

a. Are all unit tests executed every day after the build compilation?
b. Are the unit tests that developers put in the SCM system automatically included in the build without extra effort from the developers?

In a daily build, it should be possible to execute all unit tests. It must be easy for developers to add unit tests to the build. This is normally done by the developer putting their tests in the SCM system.

4. Installation

a. Is the system installed to *production-like environment* every day after the build compilation?

A *production-like environment* is the environment where a finished product should be installed. This is the hardware and environment it is running. This does not mean that it goes live automatically every day. The installation should also be completely automated.

5. Acceptance tests

a. Are all possible acceptance tests (e.g., functional/system) executed?
b. Can people easily (without much effort) add new acceptance tests?

Some acceptance tests will be running in the *production-like environment*. It is possible that some acceptance tests cannot be executed daily (e.g., because they take too long) – they will be excluded. The acceptance tests are also added to the build in a similar manner as unit tests.

6. Reporting

a. Is a failure automatically reported to the people who might have broken the build and to others?
b. Is the current status being published?

Reporting of the daily build should also be automated. A common mistake is to just mail everyone. This leads to the situation in which people ignore a failed build (because it is not their fault). Thus, reporting should be done automatically to the people who potentially broke the build (anyone who changed something since the last time it was working) and also to other interested parties such as testing or management.

7. Policy

a. Does a broken build (including tests) become the first priority for the project?

If a daily build fails then this needs to be the first priority for everybody in the project. If this is not true, then daily builds will start failing and become completely useless since they do not provide the visibility and stability in the project anymore.

CI Questions

The questions in the CI section are:

1. Anytime integration

a. Can, at any time, any developer integrate his work into the main branch without too much effort?

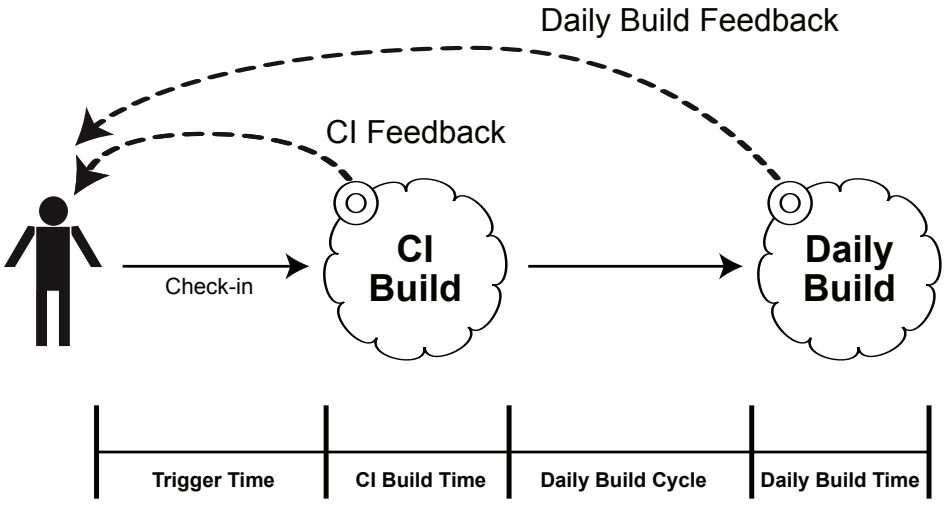
Integrate means that the check-in triggers a compile and test. The *main branch* means that it is integrated into the total product source, not on a separate feature branch.

This question is best understood by looking at what it does not mean. When there are specific integration points (e.g., three-week builds) or when developers work for weeks on subsystems or features without integrating it into the main branch, then there is not an *anytime integration* capability (the grid needs to be filled in with red).

2. Compilation

a. Within an hour after check-in, is the system automatically compiled?

Figure 1: *CI Environment*



The *anytime integration* should trigger a compilation of the software. This works on either a time-based trigger or on a trigger from the SCM system.

The time to feedback is critical in CI and, thus, in larger projects this compilation might be done incrementally or even at the subcomponent level only so that the feedback comes quickly enough.

3. Sanity check

a. Are essential tests run after the system is compiled?

The sanity check for CI is likely to be different than the one from the daily build. This sanity check consists of mainly unit tests and some acceptance tests. It would be best if the build system could find the most relevant tests automatically and execute them. Again, fast feedback is critical and the sanity check must not take too long.

4. Reporting

a. Is a failure automatically being reported to the person who integrated the work?

b. Is the current status being published?

The number of people to which a failure is reported should be smaller than with the daily build. The current status should be highly visible to the team since developers base their integration decisions on the current build status [13]. A lava lamp (using a red and green lava lamp to show the build status [14]) or a public monitor is a good solution for achieving this visibility.

5. Policy

a. Does a broken build become the first priority in the project? (This is the same as for daily build.)

TDD Question

The question in the TDD section is:

1. Are developers doing TDD?

This question is not directly related to CI but it is still included in the grid. Using TDD, a developer's private workspace always stays in a working state. When a developer increases his integration frequency, then he needs to learn to work in smaller steps. This is where TDD makes CI easier. Between the TDD cycles, a developer checks the status of the current build and check-in when the build passes. When the build fails, the developer does a few more TDD cycles and then checks in.

Metrics

The metrics used in the grid are best explained with a picture of the CI environment (see Figure 1 on page 23).

1. Integration feedback time

The integration feedback time is the time between when a developer is ready for a check-in and a CI build report. If a project is not doing *anytime integration*, then this is equal to the build feedback time. Otherwise, this is equal to the trigger time plus the maximum compile and test time.

For example, when a project uses a 10-minute trigger time, the compilation lasts five minutes, and the tests take 15 minutes, then the integration time metric would be 30 minutes. In a normal situation, a developer will know after 30 minutes if the integration has failed or succeeded.

2. Build feedback time

The build feedback time measures the build cycle. When a project has daily builds then this metric should always be one day. When a project has CI and not a daily build level then it is less than one day and the metric is not important anymore. For example, when a project makes one complete build every three weeks then the build feedback time is three weeks.

3. Test coverage

The last metric in the grid is the test coverage (during the daily build). This metric tells something about the validity of all the other fields in the grid since most of the questions do not make sense if the test coverage of the automated tests is low.

Reviews

When using CI, the effort the developer spends for integrating code needs to be minimized. The more effort that is needed for the developer, the less likely he or she is to integrate continuously, and the more likely he is going to save his work up and integrate in a bigger batch. This will seem more efficient to him but counterproductive to CI.

Minimizing the effort before integrating code often means changing reviewing practices. Formal inspections are too heavy to do when integrating multiple times a day. A quick peer-review and personal reviews might work better. Another alternative is to delay the reviewing until after the integration has been done. This way the developers can review all changes done – for example, once a day or once a week. This is easy to plan and the reviewing will be done as a shared effort which also increases the team learning.

4. Grid Example

Table 1 is an example of the grid filled in. This example contains four different projects (P1-P4). The grid starts by listing the technology and platform for each project. The build environment is often dependent on platform and technology and, thus, listing them

Table 1: *Example CI Grid*

	P1	P2	P3	P4
Technology (Program Language)	J, C++	J	J	C
Platform	Lin	Lin	Lin	Hpux
Daily Build				
Compilation				
Sanity check				
Unit testing				
Installation				
Acceptance tests				
Reporting				
Policy				
Continuous Integration				
Anytime integration				
Compilation				
Sanity check				
Reporting				
Policy				
Test-Driven Development				
Metrics (Estimations)				
Integration feedback time	24h	15m	24h	3w
Build feedback time	24h	15m	24h	3w
Test coverage	?	50-80	80	?

Legend

Not started
Working on it
We have it

makes it easier to see which projects you can share practices with.

P4 is a legacy project which does not have a daily build at all; its build cycle is three weeks. Both P1 and P3 have implemented daily compilation but neither is able to execute tests automatically yet. Therefore, their integration feedback time and build feedback time is equal. P2 is a fairly small project and they chose to only have one cycle (no daily build since everything within the CI cycle had been built). However, they only compile and execute unit tests; they do not yet have automated installation and acceptance tests.

From the grid, each project can see their potential improvement areas: P4 could start implementing daily builds, P1 and P3 could implement automated tests, and P2 could focus on automated installation. Also, from the grid we can see that the people from P3 might want to talk to the people from P2 since their environments are similar and they can learn from each other.

Conclusion

The grid offers a very simple way of measuring the CI capability of a project. It can be completed in 10 minutes and it then can provide a road map for improvement and a comparison between projects. The grid, however, limits itself to measuring the capability of the environment and not the usage of that environment by the people in the project.◆

References

1. Womack, J., and D. Jones. Lean Thinking. 2nd ed. Free Press, 2003.
2. Poppendieck, M., and T Poppendieck. Lean Software Development: An Agile Toolkit. Addison-Wesley, 2003.
3. Poppendieck, M., and T. Poppendieck. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley, 2006.
4. Ohno, T. Toyota Production System: Beyond Large-Scale Production. Productivity Press, 1988.
5. Duvall, P., S. Matyas, and A. Glover. CI: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007.
6. Cusumano, M., and R. Shelby. Microsoft Secrets. 1995.
7. McCarty, J. Dynamics of Software Development. Microsoft Press, 1995.
8. McConnell, S. Rapid Development. Microsoft Press, 1996.
9. Beck, K. Extreme Programming Explained. Addison-Wesley, 1999.
10. Fowler, M. <www.martinfowler.com/articles/continuousintegration.html>.
11. "CruiseControl." Cruise Control <http://cruisecontrol.sourceforge.net>.
12. Beck, K. Test-Driven Development. Addison-Wesley, 2003.
13. Fredrick, J. "Continuous Integration." Better Software Magazine Sept. 2004.
14. Clark, M. Pragmatic Project Automation. The Pragmatic Programmers, 2004.

About the Author

Bas Vodde is the owner of Odd-e, a small consulting company based in Singapore, that specializes in training and coaching related to Agile and Lean development. His main interests are in Scrum and especially how to use it within large companies and large projects, and also focuses on technical practices, especially TDD (including refactoring) and CI. Vodde believes you need a well-factored code base if you want to be fast and flexible. Originally from Holland, Vodde moved to China where he worked for Nokia and gained experience on large projects and the traditional ways they are run. After this, he became convinced that Agile is the way forward for all sizes of projects and moved to Helsinki, Finland to introduce Agile and watched teams adopt both Scrum and Agile practices. His interests include Lean Production, quality management, and programming, and he recently co-authored the book "Large Agile and Lean Product Development."

Odd-e Ltd. Pte
Singapore
E-mail: basv@odd-e.com