Continuous Delivery: What and How?

By Eberhard Wolff

Date: Feb 23, 2017

Sample Chapter is provided courtesy of Addison-Wesley Professional.

Return to the article

Software architect Eberhard Wolfe introduces the term Continuous Delivery and explains which problems Continuous Delivery solves and how. A first introduction into the Continuous Delivery pipeline is also given.

1.1 Introduction: What Is Continuous Delivery?

This question is not so easy to answer. The inventors of the term do not provide a real definition.[1] Martin Fowler focuses in his discussion[2] of Continuous Delivery on the fact that software can be brought into production at any time. This requires an automation of the processes necessary for the installation of software and feedback about software quality. Wikipedia[3] on the other hand defines Continuous Delivery as an optimization and automation of the software release process.

In the end, the main objective of Continuous Delivery is to analyze and optimize the process leading up to the release of software. Exactly speaking this process is often blended out during development.

1.2 Why Software Releases are So Complicated

Software releases are a challenge—very likely every IT department has already worked during a weekend to bring a software release into production. Such events often end with bringing the software somehow into production—because from a certain point the path back to the old version is even more dangerous and difficult than the path ahead. However, the installation of the release is then often followed by a long phase in which the release has to be stabilized.

1.2.1 Continuous Integration Creates Hope

Nowadays it is the release into production that represents a challenge. Not so long ago the problems started much earlier: Individual teams worked independently on their modules, and prior to the release the different versions first had to be integrated. When the modules were put together for the first time, the system frequently did not even compile. Often it took days or even weeks until all changes were integrated and compiled successfully. Only then could the deployments commence. These problems have mostly been solved by now: All teams work on a shared version of the code that is permanently automatically integrated, compiled, and tested. This approach is called Continuous Integration. The required infrastructure for Continuous Integration is detailed in Chapter 3, “Build Automation and Continuous Integration.” The fact that the former problems associated with this phase have been solved raises hopes that there will also be solutions for the problems arising during the other phases leading up to production.

1.2.2 Slow and Risky Processes

The processes in the later phases are often highly complex and elaborate. In addition, manual steps render them very tedious and error-prone. This is true for the release into production, but also for the preceding phases—for example, during testing. Especially during a manual process, which, to make things worse, is only performed a few times per year, errors are likely to occur. This of course contributes to the risk associated with the overall procedure.

Because of the high risk and complexity, releases are not very frequently brought into production. In the end this causes the processes to take even longer due to lack of practice. In addition, this makes it difficult to optimize the processes.

1.2.3 It’s Possible to be Fast

On the other hand, there are always possibilities to bring a release rapidly into production in an emergency—for instance, when an error has to be urgently repaired. However, in such a case all the tests and therefore all the safety nets, which are an integral part of the standard process, are omitted. This is of course a pretty high risk—there are good reasons to normally run those tests.

Therefore, the normal path into production is slow and risky—and in emergencies the path can be faster, but at the expense of even more risk.

1.3 Values of Continuous Delivery

Using the motivation and the approaches of Continuous Integration, we want to optimize the way for releases into production.

A fundamental principle of Continuous Integration is: “If it hurts, do it more often and bring the pain forward.” What sounds like masochism is in reality an approach for problem solving. Instead of avoiding problems with releases by bringing as few releases as possible into production, these processes should be performed as often and as early as possible in order to optimize them as quickly as possible—with regards to speed and with regards to reliability. Consequently, Continuous Delivery forces the organization to change and to adopt a new way of working.

In the end this approach is not really surprising: As mentioned, every IT organization is able to rapidly bring a fix into production—and in such a scenario it is common practice to perform only a fraction of the usual tests and security checks. This is possible because the change is small and therefore represents only a small risk. Here, another approach for minimizing risk becomes apparent: Instead of trying to safeguard against failures via complex processes and rare releases, it is also possible to more frequently bring small changes into production. This approach is in essence identical to the Continuous Integration strategy: Continuous Integration means that even small software changes by the individual developers and the team are permanently integrated instead of having the teams and developers work independently for days or weeks and integrating all the accumulated changes only at the end—a strategy that frequently causes substantial problems; in some cases the problems are so large that the software cannot be compiled at all.

However, Continuous Delivery is more than just “fast and small.” Continuous Delivery is based on different values. From these values, concrete technical measures can be deduced.

1.3.1 Regularity

Regularity means to execute processes more frequently. All processes that are necessary to bring software into production should regularly be performed—and not only when a release has to be brought into production. For example, it is necessary to build test and staging environments. The test environments can be used for acceptance or technical tests. The staging environments can be used by the final customer for testing and evaluating the features of a new release. By providing these environments, the process for the generation of an environment can turn into a regular process that is not merely performed when the production environment has to be created. To generate this multitude of environments without too much effort the processes have to become largely automated. Regularity usually leads to automation. Similar rules apply to tests: It does not make sense to postpone the necessary tests until right before the release—instead they should rather be performed regularly. Also in this case this approach basically forces automation in order to limit the necessary effort. Regularity also leads to a high degree of reliability—processes that are frequently performed can be reliably repeated and executed.

1.3.2 Traceability/Confirmability

All changes to the software that is supposed to be brought into production and to the infrastructure that is required for the release have to be traceable. It has to be possible to reconstruct each state of the software and of the infrastructure. This leads to versioning that does not only comprise the software, but also the necessary environments. Ideally, it is possible to generate each state of the software together with the environment required for the operation in the right configuration. Thereby all changes to the software and the environments can be traced. Likewise it is very easy to generate a suitable system for error analyses. And finally, changes can be documented or audited in this manner.

One possible solution for the problem is that production and staging environments are only accessible for certain persons. This is supposed to avoid “quick fixes” that are not documented and cannot be traced anymore. Besides, security requirements and data security argue against accessing production environments.

With Continuous Delivery, interventions into an environment are only possible when an installation script is changed. The changes to the scripts are traceable when they are deposited in a version control system. The developers of the scripts also do not have access to the production data so that there are also no problems with data security.

1.3.3 Regression

To minimize the risk associated with bringing software into production, the software has to be tested. Of course, the correct functioning of new features has to be assured during the testing. However, a lot of effort arises from the attempt to avoid regressions—that is, errors that are introduced by modifications in already tested software parts. This would in effect require that all tests be rerun in case of a modification since in the end a modification at one site of the system might cause an error somewhere else. This necessitates automated tests. Otherwise the required effort for the execution becomes much too high. Should an error nevertheless make its way into production, it can still be discovered by monitoring. Ideally, there is the possibility to install as simply as possible an older version on the production system without the error (rollback) or to bring a fix quickly into production (roll forward). In the end the idea is to have a kind of early warning system that takes measures throughout different phases of the project, like test and production, to discover and solve regressions.

1.4 Benefits of Continuous Delivery

Continuous Delivery offers numerous benefits. Depending on the scenario the different advantages can be of varying importance—consequently this will influence how Continuous Delivery is implemented.

1.4.1 Continuous Delivery for Time to Market

Continuous Delivery decreases the time required for bringing changes into production. This generates a substantial benefit on the business end: It becomes much easier to react to changes of the market.

However, the advantages extend beyond a faster time to market: Modern approaches like Lean Startup[4] advocate a strategy that benefits even more from the increased speed. The focus of Lean Startup is to position products on the market and to evaluate their chances at the market while investing as little effort as possible in doing so. Just like with scientific experiments, it is defined beforehand how the success of a product on the market can be measured. Then the experiment is performed, and afterwards the success or failure is measured.

1.4.2 One Example

Let us look at a concrete example. In a web shop a new feature is supposed to be created: Orders can be delivered on a defined day. As a first experiment the new feature can be advertised. Here the number of clicks on a link within the advertisement can be used as an indication for the success of this experiment. At this point no software has been developed yet—that is, the feature is not yet implemented. If the experiment did not lead to a promising result, the feature does not appear to be beneficial and other features can be prioritized instead—without much effort having been invested.

1.4.3 Implementing a Feature and Bringing It into Production

If the experiment was successful, the feature will be implemented and brought into production. Even this step can be conducted like an experiment: Metrics can help to control the success of the feature. For example, the number of orders with a fixed delivery date can be measured.

1.4.4 On to the Next Feature

The analysis of the metrics reveals that the number of orders is high enough—interestingly most orders are not sent directly to the customer, but to a third person. Additional measurements show that the ordered items are obviously birthday presents. Based on this information the feature can be expanded—for example with a birthday calendar and recommendations for suitable presents. This requires of course that such additional features are designed, implemented, brought into production and finally evaluated with regards to their success. Alternatively, there might also be options to evaluate the potential market success of these features without any implementation—via advertisements, customer interviews, surveys, or other approaches.

1.4.5 Continuous Delivery Generates Competitive Advantages

Continuous Delivery makes it possible to more rapidly bring required software changes into production. This allows enterprises to more quickly test different ideas and to develop their business model further. This creates a competitive advantage: Since more ideas can be evaluated, it is easier to filter out the right ones—and this is not based on subjective estimations of market chances, but on the basis of objectively measured data (Figure 1.1).

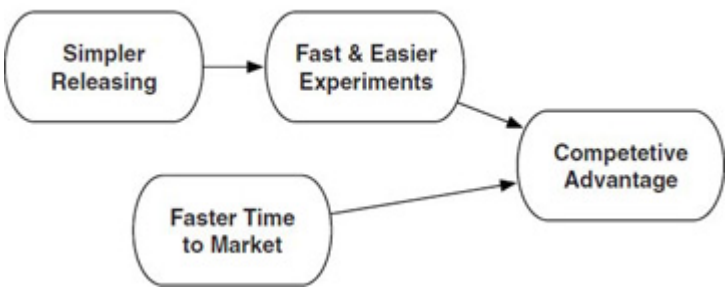

Figure 1.1 *Reasons for Continuous Delivery in a startup*

1.4.6 Without Continuous Delivery

Without Continuous Delivery the feature for the fixed delivery dates would have been planned and brought into production during the next release—this would likely have taken a number of months. Before the release, marketing would hardly have dared to advertise the feature since the long time up to the next release would render such advertisements futile. If the feature had not proven successful in the end, there would have been high costs caused by its implementation without creating any benefit. Evaluating the success of a new feature would certainly also be possible in the classical approach; however, the reaction would be drastically slower. Further developments such as the features supporting the buying of birthday presents would reach the market much later since they would require that the software be brought into production again and the time-consuming release process be run through a second time. Besides, it remains doubtful whether the success of the feature would have been analyzed in enough detail to recognize the potential for additional features supporting the shopping for birthday presents.

1.4.7 Continuous Delivery and Lean Startup

Therefore, optimization cycles can be passed through much faster thanks to Continuous Delivery because each feature can be brought into production practically at any time. This makes approaches like Lean Startup possible. This influences how the business end is working: It has to more rapidly define new features and does not have to focus on long range planning anymore, but can immediately react to the outcome of the current experiments. This is especially easy in startups, but such structures can also be built in classical organizations. The Lean Startup approach has, unfortunately, a misleading name: It is an approach where new products are positioned on the market via a series of experiments, and this approach can of course also be implemented in classical enterprises, not only in startups. It can also be used when products have to be delivered classically—for instance, on media such as CDs, with other complex installation procedures, or as part of another product such as a machine. In such a case the installation of the software has to be simplified or ideally automated. Besides a range of customers has to be identified who would like to test new software versions and be willing to provide feedback on them—that is, classical beta testers or power users.

1.4.8 Effects on the Development Process

Continuous Delivery influences the software development process: When individual features are supposed to be brought into production, the process has to support this. Some processes use iterations of one or several weeks' length. At the end of each iteration a new release with several features is brought into production. This is not an ideal approach for Continuous Delivery because in this way individual features cannot pass through the pipeline on their own. This also poses obstacles for the Lean Startup approach: When several features are rolled out at the same time, it is not obvious which change influences the measured values. Let us assume that the option for delivery on a fixed date is introduced in parallel with a change of the shipment costs—it will not be possible to distinguish which of the two changes had a greater influence on the higher number of sold items.

Therefore, processes like Scrum, XP (Extreme Programming), and of course the waterfall are impedimentary since they always bring several features together into production. Kanban,[5] on the other hand, focuses on bringing a single feature through the different phases into production. This fits ideally with Continuous Delivery. Of course, the other processes can also be modified in ways that allow them to support the delivery of individual features. However, in such a case the processes have been adapted and are not implemented according to the textbook anymore. Another possibility is to initially deactivate the additional features in order to bring several features together in one release into production, but still be able to measure their effects separately.

In the end this approach also means that teams include multiple different roles. In addition to development and operation of the features, business roles such as marketing are conceivable. Thanks to the decreased organizational hurdles, the feedback from the business end can be translated into experiments even faster.

Try and Experiment

- Gather information about Lean Startup and Kanban. Where did Kanban come from originally?

  Choose a project you know or a feature in a project:

- What could a minimal product look like? The minimal product should give an idea about the market chances of the planned complete product.
- Is it also possible to evaluate the product without software? Is it, for instance, possible to advertise it? Are interviews of potential users an option?
- How can the success of the feature be measured? Is there, for instance, an influence on sales, a number of clicks, or another value that could be measured?
- How much time in advance do marketing and sales typically have for planning a product or feature? How does that fit to the Lean Startup idea?

1.4.9 Continuous Delivery to Minimize Risk

The use of Continuous Delivery as described in the last section goes together with a certain business model. However, for classical enterprises the business often depends on long-range planning. In such a case an approach like Lean Startup cannot be implemented. In addition, there are many enterprises for which time to market is not a decisive factor. Not all markets are very competitive in this regard. This can of course change when such companies are suddenly confronted with competitors that are able to enter the market with a Lean Startup model.

In many scenarios time to market cannot motivate the introduction of Continuous Delivery. Still the techniques can be useful since Continuous Delivery offers additional benefits:

- Manual release processes require a lot of effort. It is no rare event that entire IT departments are blocked for a whole weekend for a release. And after a release there is frequently still extensive follow-up work to do.

- Also the risk is high: The software rollout depends on many manual modifications, which easily leads to mistakes. If the errors are not discovered and fixed in time, this can have far-reaching consequences for the enterprise.

The sufferers are found in the IT departments: Developers and system administrators who work through weekends and nights to bring releases into production and to fix errors. In addition to working long hours they are subjected to high stress because of the high risk. And the risks should not be underestimated: Knight Capital, for instance, lost $440M because of a failed software rollout.[6] As a consequence the company went into insolvency. A number of questions[7] arise from such scenarios—in particular why the problem occurred, why it wasn't noticed in a timely manner, and ultimately how such events can be prevented in other environments.

Continuous Delivery can be a solution for such situations: Fundamental aspects of Continuous Delivery are the higher reliability and the quality of the release process. This allows developers and system administrators to sleep calmly in the true sense of the word. Different factors are relevant for this:

- Due to the higher level of automation of the release processes, the results become easier to reproduce. Thus, when the software has been deployed and tested in a test or staging environment, the exact same result will be obtained in production because the environment is completely identical. This allows largely eliminating sources of error, and consequently the risk decreases.
- In addition, testing software becomes much easier since the tests are largely automated. This increases the quality further as the tests can be performed more frequently.
- When there are more frequent deployments, the risk decreases likewise since fewer changes are brought into production per deployment. Fewer changes translate into a smaller risk that an error has crept in.

In a way, the situation is paradoxical: The classical IT tries to bring releases as seldom as possible into production since they are associated with a high risk. During each release an error with potentially disastrous consequences can creep in. Fewer releases should therefore result in fewer problems.

Continuous Delivery on the other hand advocates frequent releases. In that case fewer changes go live at each release, which also decreases the probability for the occurrence of errors. Automated and reliable processes are a prerequisite for this strategy. Otherwise the frequent releases lead to an overload of the IT personnel performing manual processes, and in addition the risk increases since errors are more prone to occur during manual processes. Instead of aiming at a low release frequency the relevant process are automated to decrease the release-associated risk. It is of course an added advantage that in case of a high release frequency the individual releases comprise fewer changes so that the inherent risk of errors is lower.

Here, the motivation for Continuous Delivery (Figure 1.2) thus profoundly differs from that of the Lean Startup idea: The focus is on reliability and a better technical quality of the releases—not on time to market. And the beneficiaries are the IT departments—not only the business domains.

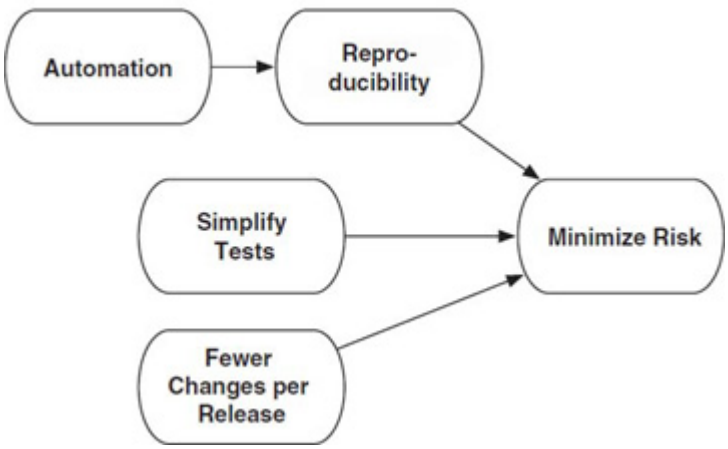

Figure 1.2 *Reasons for Continuous Delivery in an enterprise*

Since the benefits are different, other compromises can be made: For example, investing in a Continuous Delivery pipeline is often worthwhile even if it does not extend all the way up to production—that is, when the production environment still has to be built manually. In the end the production has only to be built once for each release, but multiple environments are required for the different tests. However, if time to market is the main motivation for Continuous Delivery, it is essential that the pipeline include production.

Try and Experiment

Look at your current project:

- Where do problems typically arise during installation?

- Could these problems be solved by automation?
- Where should the current approaches be simplified in order to facilitate automation and optimization? Evaluate the required effort and the expected benefit.
- How are production systems and test systems currently built? By the same team? Would it be conceivable to apply automation to both areas or only to one of the two?
- For which systems would automation be useful? How often are the systems built?

1.4.10 Faster Feedback and Lean

When a developer modifies the code, she receives feedback from her own tests, integration tests, performance tests, and finally from production. If changes are brought into production only once per quarter, several months can pass between the code modifications and the feedback from production. The same can hold true for acceptance or performance tests. If an error occurs then, the developer has to think back to what it was she had implemented months ago and what the problem might be.

With Continuous Delivery the feedback cycles become much faster: Every time the code passes through the pipeline the developer and his/her entire team receive feedback. Automated acceptance and capacity tests can be run after each change. This enables the developer and the development team to recognize and fix errors much more rapidly. The speed of feedback can be further increased by preferring fast tests, such as unit tests, and by first testing broadly and only afterwards testing deeply. This ensures from the start that all features function at least for easy cases—the so-called “happy path.” This makes spotting basic errors easier and faster. In addition, tests that are known from experience to fail more often should be executed at the start.

Continuous Delivery is also in line with Lean thinking. Lean regards everything that is not paid for by the customer as waste. Any change to the code is waste until it is brought into production since only then will the customer be willing to pay for the modifications. Besides, Continuous Delivery implements shorter cycle times for faster feedback—another Lean concept.

Try and Experiment

Have a look at your current project:

- How much time passes between a code change and
  - feedback from a Continuous Integration server?
  - feedback from an acceptance test?
  - feedback from a performance/capacity test?
  - bringing it into production?

1.5 Generations and Structure of a Continuous Delivery Pipeline

As already mentioned, Continuous Delivery extends the approach of Continuous Integration to additional phases. Figure 1.3 offers an overview of the phases.

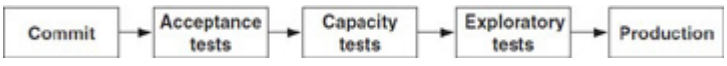

Figure 1.3 *Phases of a Continuous Delivery pipeline*

This section introduces the structure of a Continuous Delivery environment. It is oriented along Humble et al. (see footnote 1) and consists of the following phases:

- The commit phase comprises the activities that are typically covered by a Continuous Integration infrastructure such as the build process, unit tests, and static code analysis. Chapter 3 discusses this part of the pipeline in detail.
- Chapter 4, “Acceptance Tests.” Strictly speaking, the topic is automated tests: Either the interactions with the GUI are automated to test the system or the requirements are described in natural language in a manner that allows them to be used as automated tests. From this phase on, if not before it is necessary to generate environments on which the applications can run. Therefore, Chapter 2, “Providing Infrastructure,” deals with the question of how such environments can be generated.

- Capacity tests (Chapter 5, “Capacity Tests”) ensure that the software can cope with the expected load. For this purpose an automated test should be used that unambiguously indicates whether the software is sufficiently fast. The relevant point is not only performance, but also scalability. Therefore, the test can also take place in an environment that does not correspond to the production environment. However, the environment has to be able to deliver reliable results about the behavior in production. Depending on the concrete use case other non-functional requirements, such as security, can also be tested in an automated fashion.

- During explorative tests (Chapter 6, “Exploratory Testing”) the application is not examined based on a strict test plan. Instead, domain experts test the application with a focus on new features and unanticipated behaviors. Thus, even in Continuous Delivery not all tests have to be automated. In fact, by having a large number of automated tests, capacity is freed for explorative testing since routine tests do not have to be manually worked off anymore.

- The deployment into production (Chapter 7, “Deploy—The Rollout in Production”) merely comprises the installation of the application in another environment and is therefore relatively low risk. There are different approaches to further minimize the risks associated with the introduction into production.

- During operation of the application, challenges arise—especially in the areas of monitoring and surveillance of log files. These challenges are discussed in Chapter 8, “Operations.”

In principle, releases are promoted into the individual phases. It is conceivable that a release manages to reach the acceptance test phase and successfully passes the tests there, but shows too low a performance during the capacity tests. In such a case the release is never going to be promoted into the following phases, like explorative testing or production. In this manner, the software has to show that it fulfills increasing requirements before it finally goes into production.

Let us assume for example that the software contains an error in the logic. Such an error would at the latest be discovered during the acceptance tests, since those check the correct implementation of the application. As a consequence, the pipeline would be broken off (Figure 1.4). Additional tests are not needed anymore at this point.

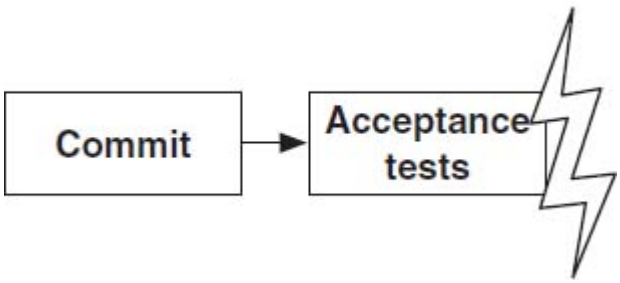

Figure 1.4 *Continuous Delivery pipeline stops at acceptance tests*

The developers will then fix the error, and the software is built anew. This time it also passes the acceptance test. However, there is still an error in a new function for which there is no automated acceptance test. This error can only be discovered during the explorative tests. Consequently, this time the pipeline is interrupted at the explorative tests, and the software does not go into production (Figure 1.5). This prevents testers wasting time with software that does not fulfill the requirements with regards to load handling, or that contains errors that can be detected by automated tests.

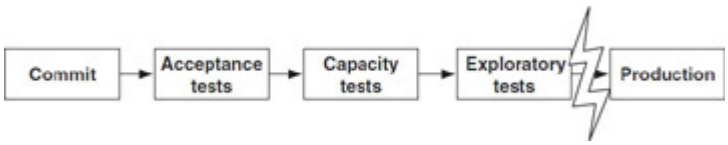

Figure 1.5 *Continuous Delivery pipeline stops at explorative tests*

In principle, several releases can be processed in the pipeline in parallel. Of course this requires that the pipeline support multiple releases in parallel—if the tests are running in fixed environments, this is not possible since the environment will be occupied by a test so that a parallel test of a second release cannot run at the same time.

However, it is very rare that releases are processed in parallel by Continuous Delivery. A project should have exactly one state in the version administration, which is then promoted through the pipeline. At the most it might happen that modifications to the software occur with such a speed that a new release is already sent into the pipeline before the previous release has left the pipeline. Maybe there are exceptions for hotfixes—but one objective of Continuous Delivery is just to treat all releases equally.

1.5.1 The Example

section P.2). This example is intentionally kept very simple concerning the domain logic. Essentially the first name, name, and email address of a customer are registered. The registrations are validated. The email address has to be syntactically

correct, and there is only one registration allowed per email address. In addition, a registration can be searched based on the email address, and can be deleted.

Since the application is not very complex, it is relatively easy to understand so that the reader can concentrate on the different aspects of Continuous Delivery that are illustrated by the example application.

Technically the application is implemented with Java and the framework Spring Boot. This makes it possible to start the application, including web interface, without installing a web or application server. Thus the testing becomes easier since no infrastructure has to be installed. However, the application can also be run in an application or web server like Apache Tomcat if that is necessary. The data are stored in HSQLDB. This is an in-memory database that runs inside the Java process. This measure also reduces the technical complexity of the application.

The source code of the example can be downloaded at http://github.com/ewolff/user-registration-V2. An important note: The example code contains services that run under root rights and can be accessed via the net. This is not acceptable for production environments because of the resulting security problems. However, the example code is only meant for experimenting. For that the easy structure of the examples is very useful.

1.6 Conclusion

Putting software into production is slow and risky. Optimizing this process has the potential to make software development overall more effective and efficient. Continuous Delivery might therefore be one of the best options to improve software projects.

Continuous Delivery aims at regular, reproducible processes to deliver software—much like Continuous Integration does to integrate all changes. While Continuous Delivery seems like a great option to decrease time to market it actually has much more to offer: It is an approach to minimizing risk in a software development project because it ensures that software can actually be deployed and run in production. So any project can gain some advantage—even if it is not in a very competitive market where time to market is not that important after all.
