SMHARTER

TURN YOUR WAY OF WORKING INTO
YOUR COMPETITIVE ADVANTAGE

Software Development Done Right: The Foundations a CTO Cannot Ignore

This short booklet introduces a minimum set of practices for software development and IT departments that you should never neglect in professional software development. You'll find also references to conclusive scientific research studies, industry opinion, suggested reading material, and suggested actions to improve in those areas.

Table of Contents

Everyday, CTOs make IT decisions and chart strategic directions in turbulent and uncertain circumstances:

- New technologies emerge every three to six months and a new paradigm shift happens every year or two.
- Every new generation of IT professional wants to start from scratch, ignoring the past and creating new trends.

What yesterday was true, right, and good, today may seem wrong, disputable, or controversial. Which certainties can a CTO rely on?

The following sections document fundamental building blocks of software development done right: a list of practices whose effectiveness has been scientifically demonstrated, practices generally accepted by the industry as suggested, trending, and cutting edge.

Each section, for each practice, references conclusive scientific research, industry opinions, suggested reading material, information for assessing if practice is properly in place or not, a short description of what good looks like, and suggestions for next actions.

Quoted scientific studies and experiments and the conclusiveness of evidence are based on work published in this book, highly recommend to CTOs:

[1] *Making Software: What Really Works, and Why We Believe It.* Andy Oram and Greg Wilson, eds. (O'Reilly Media, 2010).

Generally accepted practices in the IT industry are extracted from:

[2] *Facts and Fallacies of Software Engineering.* Robert Glass (Addison-Wesley Professional, 2002)
[3] "The Joel Test: 12 Steps to Better Code". Joel Spolsky (Joel on Software, 2000)

But first, here are a few words about why the fundamental building blocks of software development done right are relevant and have a huge impact.

Good decision making and 10x variation in productivity

Decisions that CTOs make, on technologies, hiring, and ways of working, can have a 10x impact on productivity, for better or for worse. This is the potential difference between good decisions and bad decisions.

Evidence from research studies and empirical experiments support the general claim of order-of-magnitude differences (from 5x to 28x) in productivity and quality among different programmers and among different teams (from 3x to 10x), and this applies to organisations too.

An example that supports the previous claim is a study based on data released by Lotus and Microsoft on the development of two similar products: Lotus 123 version 3 and Microsoft Excel 3.0. The Excel team members were 8x more productive than Lotus team members.

There is also plenty of anecdotal support, for example from Boeing Company in the mid-1980s, when a single programmer replaced a team of 80 and completed the delivery on time.

I also directly observed the difference in productivity in June 2015 as part of a due diligence. I observed two organisations develop and support similar products to serve similar markets with similar numbers of customers. One organisation was able to get the job done with 30 developers while the other employed 300 developers. The first organisation produced better internal quality and better quality in use of the product. My observation supports the claim of 10x variation in productivity. (For confidentiality reasons, I cannot mention the companies involved.)

The following paragraphs on software-development practices can test the effectiveness of a CTO's decision making since nine out of ten fall completely within the authority and responsibility of a CTO.

Suggested readings:

[1] "Chapter 30: What Does 10x Mean? Measuring Variations in Programmer Productivity"
[2] Fact 1 and Fact 2 in "Chapter 1: About Management"

Research studies and supporting evidence:

All research studies listed in [1] can be found below in Appendix 1.

Software design

This section discusses the first two fundamental building blocks of software development that should always be on the CTO's radar.

These basic building blocks represent a minimum set of practices for software development and IT departments.

They are **basic** in the sense that ignoring them should be considered professional negligence and in the sense that they all are necessary, but not sufficient.

They represent the **minimum** in the sense that the list only includes practices that conclusive scientific research has proven true and practices that are currently accepted as industry standards. Trending or cutting-edge practices, whose validity is still debatable, are explicitly marked as such to avoid confusion.

While some of the practices presented here and in the coming paragraphs may seem obvious, most organisations do not adopt them either properly or at all. Most common mistakes include:

- failing to translate correctly the theory into practice;
- focusing on secondary aspects while missing the fundamentals;
- applying over-engineered and overly complicated solutions; and
- overestimating the importance of tools and ignoring the importance of skills, discipline, and social aspects.

1) Software design: Effectiveness of modularisation and design patterns

Scientific research

A study on the effectiveness of modularisation, whose analysis included three large open-source projects, has shown that current modularisation techniques make developers work more effectively because the techniques reduce the time needed to make changes (see chapter 20 in [1]).

Three research studies, which included experiments with professional programmers, have shown strong empirical evidence that design patterns and design principles together can help improve software quality and productivity in a maintenance context (see chapter 21 in [1]). The positive effect during initial development has yet to be researched.

The scientifically demonstrated conclusion is that good design effectively increases productivity and quality.

The list of conclusive studies on this topic in [1] is available below in Appendix 2.

Industry-recommended practice

Good modularisation and use of design patterns in addition to basic design principles are industry-recommended practices because they have a positive impact on productivity and quality (see Fact 20 and Fact 52 from [2]).

Are you doing it?

A careful inspection of the source code, design, architecture, and repository history can produce a definitive answer to this question. A faster and indirect way of assessing the situation is to look for these clues:

- Is it increasingly time-consuming and difficult to make changes to the code and add new features?
- Is it difficult to add new features or to implement feature changes without adding new bugs to existing parts of the system?
- Do new team members require a substantial amount of time to become familiar with the code base and become productive, and is that time constantly increasing?

If you observe two or more of these problems, it's likely that you do not have good design practices properly in place. Ask yourself: do developers have the time to practice good design? Do they have the experience and skills to do that? Do business people and managers understand the business impact of good design and act accordingly?

What good looks like

Where good design modularisation is in place and basic design principles and design pattern are properly adopted:

- The effort required to add new features and make changes decreases over time.
- Code is easy to understand.
- Implementation of a single change request requires changes to a limited number of modules.
- There are only few dependencies between modules.
- Each module depends only on a limited number of other modules.

As a consequence, the defects in the code tend to be fewer and under control.

Next actions

Here are a few things you can do to improve this practice in your organisation:

- You can ask your senior developers and tech leads to work together to assess the current state of the code base in relation to modularisation and design principles and patterns, and to produce a list of recommendations.
  You can also ask them to organise internal training for developers.
  You can facilitate a dialogue between your tech leaders, business people, and managers to explore the business consequences of design practice, share the current status in different products code-base, and explore various tactical and strategic options.

How we can help

- We can assess your code base and software-engineering practices with the involvement of your tech leaders to produce a list of recommendations and an action plan.
- We can deliver one of our most successful training courses to train your developers: Agile Object-Oriented Bootcamp.
- We can run a session with your tech and business leaders about the business impact of technical decisions and facilitate a meeting to explore the way forward.

Do you want to know more on the subject?

Here are some suggested readings:

- *Refactoring to Patterns* by Joshua Kerievsky
- *Clean Code: A Handbook of Agile Software Craftsmanship* by Robert C. Martin
- *Refactoring: Improving the Design of Existing Code* by Martin Fowler et al.
- *Agile Software Development: Principles, Patterns, and Practices* by Robert C. Martin
- *Agile Principles, Patterns, and Practices in C#* by Robert C. Martin
- *Patterns of Enterprise Application Architecture* by Martin Fowler

2) Organisations and team structure: Conway’s corollary

Scientific research

Various research findings conclude that organising people (teams, departments, and physical location) in a software effort in a way that resembles the organisation and structure of the software (modularisation) produces a better result than when they differ. This is because it ensures a strong relationship between social and software structures. The definition of “better” here is:

- sustaining high levels of productivity while
- maintaining software quality.

Two of these research studies looked at commercial software development, one of which was Microsoft Windows Vista. Five analysed large open-source projects such as PostgreSQL and Apache.

The list of research studies on this topic taken from [1] is available in Appendix 3 below.

Industry-recommended practice

Current industry-recommended practice for aligning the organisation of people with software structure is to organise or re-organise both software structure and people around a product.

This means having at least the IT team fully dedicated to a product, and the code base organised primarily around that product.

Preferably, product managers, experts, and business people are also organised around the product, together with the IT people.

From a technical and code-base point of view, the most popular approaches to dealing with commonalities among different products are 1) a shared platform or 2) an internal open-source model (each product team makes available parts of its code base to other internal product teams that can reuse them and contribute to them).

Some of the well-known and successful industry examples of the practice described in this paragraph are Amazon’s two-pizza teams, Apple product teams, Spotify's tribes, Valve’s cabal system, Scrum’s team structure, and LeSS (large-scale scrum) team structure.

In the past, the aim to reduce costs often lead to outsourcing and de-localisation, misaligning the organisation of people and the software structure. The negative effects of this approach often created additional costs that largely outweighed the promised savings.

Nowadays, even IT-outsourcing pioneers GE and GM are taking software development back in-house. Jim Fowler, former CIO of GE Power & Water and now GE Capital CIO, says that GE is getting better-quality code and higher throughput using internal staff than it did with contract resources. Fowler sees that in measures such as fewer tickets for rework on post-production code built by GE-badged employees. He doesn't blame GE’s software-development partners for that difference in quality but sees it as a natural outcome of small and agile employee teams better understanding business needs, and each employee knowing that they will be the person who will be there in the long term to deal with any problems (InformationWeek).

GM CIO Randy Mott has said, “IT has become more pervasive in our business and we now consider it a big source of competitive advantage” (The Economist).

Are you doing it?

If you have stable, cross-functional, multidisciplinary teams, each built around a single product and each of which includes all people required for the ideation, development, operation, and maintenance of the product, and you have just few loosely coupled dependencies between different teams then yes, you are doing it.

To clarify, functional teams and departments (i.e., having a QA team, a BA team, a PM team, etc.), component teams, and matrix organisations all substantially violate this practice.

Do you have many cross-team dependencies? Do these dependencies and different priorities of different teams make it difficult to plan for a product? Are dependencies causing lot of overhead in planning and execution? Can emergencies or delays faced by one team cause a huge ripple effect on other product teams? Do cross-team dependencies make it difficult to test a product and declare it done, and is it difficult to plan its release into production?

If you answered yes to two or more of these questions then no, you are not doing it properly.

What good looks like

Teams and architecture are organised around a product, team membership is stable (i.e., members don't change teams for 18 months), and team members are all at least in the same time zone and preferably colocated. Special attention and effort is made to avoid and reduce cross-team dependencies.

All product teams of the company align around common goals and strategic directions and are capable of helping each other and spotting possible synergies while maintaining a level of autonomy that enables speed.

Next actions

Here few things you can do to improve this practice in your organisation:

- Ask all employees working on a product to draw a value-stream map of the product delivery from concept to cash and analyse delays and problems where work is handed over to different teams and departments.
- Experiment with a small cross-functional team when you have the opportunity to deliver a new product.

How we can help:

- Our agility analysis can assess your status, identify strengths and pain points, and recommend suggested improvements and options for implementing them.
- We can deliver a half-day Organisational Agility for Executives workshop to decision makers and representatives from different functions to explore possibilities and ways to increase the alignment of people organisation and software structure around products.

We can facilitate a meeting with leaders and managers involved in a specific software effort and facilitate a discussion on a better alignment around the product.

Do you want to know more on the subject?

Here are some suggested readings:

- *Agile IT Organisation Design: For Digital Transformation and Continuous Delivery* by Sriram Narayan
- *Lean Enterprise: How High Performance Organizations Innovate at Scale* by Jez Humble, Joanne Molesky, and Barry O'Reilly
- “Exploring the Duality Between Product and Organizational Architectures: A Test of the ‘Mirroring’ Hypothesis” by Alan MacCormack, John Rusnak, and Carliss Baldwin (Harvard Business School working paper)
- “Product-Centric Development Is a Hot New Trend” by Dave West et al. (Forrester report)
- “Maverick Research: Lessons Learned From Case Studies for Ultralean Development” by Ray Valdes (Gartner report)
- “Technical Dependency Challenges in Large-Scale Agile Software Development” by Nelson Sekitoleko et al. (in *Agile Processes in Software Engineering and Extreme Programming*)

Version-control systems and build automation

This section introduces the third and fourth fundamental building blocks of software development done right, which the CTO needs to be aware of.

The vast majority of organisations that do professional software development implements these two rules. In a sense, these two practices mark the distinction between novice/amateur software development and professionals.

The benefit of these practices is so evident that you won’t find any scientific research done to prove the obvious. Indeed, they are the first two points Spolsky makes in [3].

On the other hand, many organisations don't have a good understanding of what good looks like for these two practices and sometimes adopt over-engineered solutions while missing key aspects.

3) Source-code repository a.k.a. a version-control system

Industry-recommended practice

Use of a source-code repository (e.g., SVN, Git, Visual SourceSafe, Rational ClearCase, Mercurial, Perforce, TFS) is a standard practice to support collaboration among developers working on a shared code base and to ensure that no code gets lost.

Are you doing it?

You are doing it when:

- You store all production source code that you create in a source-code repository.
- You build all binaries only from code in the source-code repository.
- Developers check out (get or pull) code at least once per day before starting to work and check in (commit or push) code at least once per day before the end of the day.

What good looks like

This is where most mistakes are found.

A source-code repository should be auto-contained. That means that all source code, configuration files, and database schema files required to build an application are contained in one single repository and a build neither requires nor references any code from other repositories.

In the best organisations, developers check out and check in code many times per day, typically every 15 minutes or every hour. The most effective way to deal with merge conflicts is to avoid them. Frequent check-ins and check-outs have many other benefits.

When using distributed version-control systems (e.g., Git, BitKeeper, Mercurial, Bazaar, etc.) to work with remote teams or remote team members or when simply using a source-code repository hosted in the cloud, make sure that everyone can continue to work in the event of an interrupted connection to the repository. This is one of the main reasons to have a distributed version-control system instead of a traditional version-control system; if you are not taking advantage of it, you should wonder why you are using a distributed system and paying the price of its additional complexity.

It's common and convenient to use a separate repository for binaries (e.g., JFrog's Artifactory, Sonatype's Nexus) to store third-party binaries which must be treated differently and for binaries an organization itself produces such as releases and nightly builds.

Next actions

- Ask tech leads and teams to review the status of all source-code repositories in the organisation and to verify whether they are auto-contained or not, and ask them to suggest ways to reorganise them in order to make them auto-contained.
- If you are using a distributed version-control system, simulate an interruption in the connection to the remote repository and verify that all teams are able to continue to work normally, get and push code on the local repository, and do all the related work (e.g., integrate, test, release).
- Ask tech leads and teams to review the frequency of commits/check-ins in the source-code repository. Ask them what can be done to ensure that everyone performs commits/check-ins at least once per day and what can be done to help developers to check in more frequently, ideally every 15 minutes.

How we can help:

- We can analyse your engineering and coding practices to assess your status, identify strengths and pain points, and recommend improvements and options for implementing them.
- We can deliver the Agile Development Practices training session to help developers work in short stints with modern approaches.

4) Build automation

Industry-recommended practice

A build of a software product is usually automated with a batch script. An automated build runs as a single step that doesn’t require any manual intervention.

The script does a full check-out from scratch of the latest source-code snapshot, and rebuilds every line of code. The script optionally creates a deployment or installation package. A dedicated build server is used to create release candidates.

The automatic build script can typically run both on a developer machine and on the build server. The automatic build is run at least once per day, for example at night in the build server.

Are you doing it?

You are doing it when you have daily or nightly builds automatically created in the build server and when developers can get all the code they need to work on a new computer by simply running the build script on the machine.

The automatic build script is versioned in the source-code repository and works in one step, without the need for manual intervention or tweaking. It only needs a connection to the source-code repository, the repository client tool, and local installation of tools required to build the source code.

What good looks like

- The automatic build script is stable and run smoothly on different machines, for many different users and on different build servers.
- The automatic build is fast and provides clear error messages that help identify the source of a problem.
- The automatic build is run for every developer's check-in, typically many times per day.

Next actions

- Ask new developers that join the organisation how difficult or easy it was to set up the local environment and run the build.
- Rotate the responsibility to support the automatic build and ask how difficult the handover is and how much time is required to support the automatic build.
- Ask IT operations and developers to review the infrastructure (build servers, version-control-system servers, file servers, developers' machines, networks, accounts, etc.) and to identify and remove complexities that make the automatic build more fragile and less stable.

How we can help:

- We can analyse your engineering and coding practices to assess current status, identify strengths and pain points, and recommend improvements and options for implementing them.
  We can deliver the Introduction to Continuous Delivery training session that, among other things, provides an overview of build automation.

Do you want to know more on these subjects?

Here are some suggested readings:

- *Continuous Integration: Improving Software Quality and Reducing Risk* by Paul M. Duvall, Steve Matyas, and Andrew Glover
- "Are you taking advantage of your Distributed Version Control System?" by Luca Minudel (blog post)

Iterative development

Iterative software development is the fifth fundamental building block of software development done right and should always be on the CTO's radar.

Iterative development is a stepping stone to many modern practices. Like the previous two practices, it also marks the distinction between novice/amateur and professional software development.

5) Iterative development

Industry-recommended practice

Iterative development first appeared as an experiment in 1957. In the 1990s, it became common and a CHAOS report in 1998 contained statistics from the industry that showed the advantages of Iterative over waterfall development. In 2000, the US Department of Defence stated a clear preference for Iterative development.

Iterative development was a stepping stone on the way to modern practices such as time-boxed iterations/sprints (e.g., XP and Scrum), pull systems and continuous flow (e.g., lean software development and kanban), and continuous delivery.

Waterfall and its evolutions and extensions such as V-model development are inefficient in the vast majority of circumstances and should be considered obsolete. On the other end of the spectrum, "just do it" approaches are inadequate for organisations with more than three employees. Adopting one of these approaches should be considered professional negligence, unless the exception is justified by empirical data (e.g., up-front estimation errors of less than 10% for 90% of estimates, users enthusiastically accept released features, quality is good, and software-development performance including integration and testing is at least as efficient as the competitors'). The burden of proof falls on the proponent of the exception.

Are you doing it?

Iterative development is the development or evolution and maintenance of a new software product through a sequence of iterations. Every iteration includes activities such as requirements analysis, design, implementation, and testing. Each iteration delivers new functionality or evolves existing functionalities and incorporates lessons from previous iterations.

Iterative development in its simplest form can have iterations that differ in duration. The practice can also have activities that relate to a single feature (such as requirements analysis, design, implementation, or testing) that span multiple consecutive iterations.

What good looks like

Iteration length is fixed for every iteration and varies from one month to one week — and the shorter, the better.

Every iteration is self-contained, in the sense that all activities required to add or evolve a feature, from detailed analysis of requirements to testing, are carried out and completed during the iteration, including acceptance of the feature by the business stakeholders, requester, or users.

The best teams are capable of releasing completed features into production after every iteration. The business can decide to release a feature or not, but if business decides to release it, it can enter production with one click, because features completed after every iteration are really ready to be released.

The best teams, once they achieve the ability to successfully run one-week-long iterations, can move to continuous-flow development or pull systems such as lean software development and kanban instead of moving to shorter iterations. Teams that practice continuous-flow development without being able to successfully practice iterative development in one-week iterations don't belong to this category.

Next actions

Ask team members, stakeholders, and managers to individually draft the process from requirement analysis to release. Observe collectively the differences among processes designed by different people and among different iteration goals (bug fixing, feature change or evolution, new feature for a new product, emergency bug fix, etc.) and discuss which differences are beneficial and which can be eliminated. Observe and note similarities, and discuss how well those elements of the process serve their purpose.

How we can help:

- We can analyse your way of working to assess current status, identify strengths and pain points, and recommend improvements and options for implementing them. We can deliver coaching or training based on the result of the assessment.

Test and delivery automation and continuous integration: Current trends

This section introduces three trending practices in software development that should always be on the CTO's radar.

These practices are implemented by many organisations, especially by the most successful and innovative ones. Most respected and recognized professionals in software development consider these practices and related skills essential.

Unlike for previous items in this booklet, no scientific studies have proven the effectiveness of these practices compared to alternatives; nor are they implemented by the vast majority of the organisations that do professional software development.

For these reasons, these practices cannot be considered as fundamental building blocks of software development, and failure to adopt these practices cannot be considered a CTO's professional negligence. Doing that would be confusing popularity with scientific evidence and craftsmanship.

A CTO, in regard to these practices, has the responsibility to assess (explore these practices with the goal of understanding their potential impact on the organisation) and run trials (understand how to build the capability to adopt those practices and experiment with them where risk is acceptable). Failure to do this may be considered professional negligence.

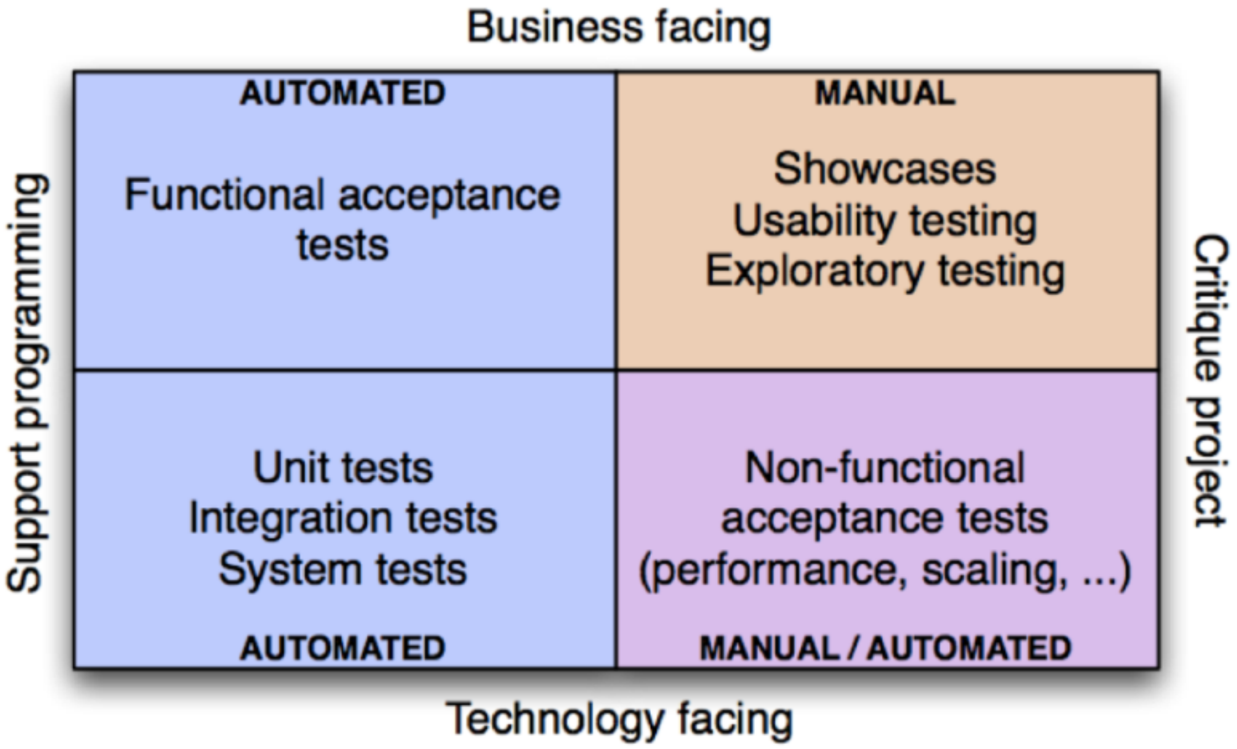
6) Test automation

Industry-trending practice

The current industry trend is to invest in test automation, including acceptance testing, integration testing, and unit testing — that's test-driven development. The industry's motivation to adopt and practice test automation is to reduce the time required by repetitive manual testing and to reduce the human error in manual testing tasks.

If your competitor has fewer defects and fewer incidents in production while releasing new features more quickly and more often, it's likely that your competitor has found a way to exploit automatic testing. It's very likely that throwing more testers, cutting regression testing, or reducing the frequency of releases won't help you keep up.

The following picture, inspired by Brian Marick's test matrix, classifies different types of testing and shows which you can automate. As you see, there's great potential in test automation.



Business facing

Support programming

Critique project

AUTOMATED

Functional acceptance tests

MANUAL

Showcases
Usability testing
Exploratory testing

Unit tests
Integration tests
System tests

AUTOMATED

Non-functional acceptance tests (performance, scaling, ...)

MANUAL / AUTOMATED

Technology facing

Are you doing it?

When you are automating tests, you have a suite of unit, integration, and acceptance tests that automatically run and report the results in few minutes. The suite takes care of the setup of the initial conditions and of the verification of the results. No manual intervention is needed.

The test suite runs after every build, which is triggered by changes pushed by a developer to the source-code repository.

Creation of deployment packages for candidate releases and releases into production are subordinate to a positive result in the automatic tests.

7) Deployment automation

Industry-trending practice

Similar to build automation, deployment automation can be implemented with a batch script that automates the deployment of the software system in test, staging, and production environments.

The script runs in a single step and requires no manual intervention. The system update is automatically delivered with all the required changes including, for example, database schema changes and configuration changes.

The same script is used for all environments so that it gets implicitly tested in test and staging environments before being used for the production environment.

Are you doing it?

You are doing deployment automation when the deployment package created by your build includes the right version of the automated deployment script and when developers and testers and IT operations can deploy any version of the software on their computers or to any test and staging environment simply by running the deployment script.

The automatic deployment script is versioned in the source-code repository, and it simply works in one step, without need for manual intervention or tweaking to make it run. It only needs access to the local system, to the binaries included in the deployment package, and the minimum privileges required to make the changes needed to the system.

When the deployment script execution ends, the system is up and running without any manual intervention or tweaking.

Similar to an automated build, automatic deployment runs at least once per day in test or staging environments.

What good looks like

In the best organisations, the automatic deployment runs in the automatic-test environment after every developer’s check-in as part of the continuous-integration process, typically many times per day.

When a deployment fails, the automatic-deployment script leaves the system in a known state. Ideally, the deployment either succeeds or rolls back to the previous version; in any case, it never leaves the system in an unknown or not-working state.

The architecture of the system and the automatic-deployment script are conceived in a way that the deployment does not interrupt the work of users using the system at the moment of the update.

8) Continuous integration

Industry-trending practice

Martin Fowler in part defined continuous integration as a software-development practice in which team members frequently integrate their work, at least daily. An automated build, including automated tests, verifies each integration to detect integration errors as quickly as possible. Fowler first posted about continuous integration in 2000, nine years after Grady Booch first named the practice in 1991.

Continuous integration is often confused with having a continuous-integration server tool — but the important part of continuous integration is to frequently, at least daily, merge the code on the mainline (also known as the head or trunk) of the source-code repository and quickly fix builds and tests that fail.

Are you doing it?

Per the definition, you are using continuous integration when all developers frequently integrate their own work and an automated build that includes automated tests verifies each integration.

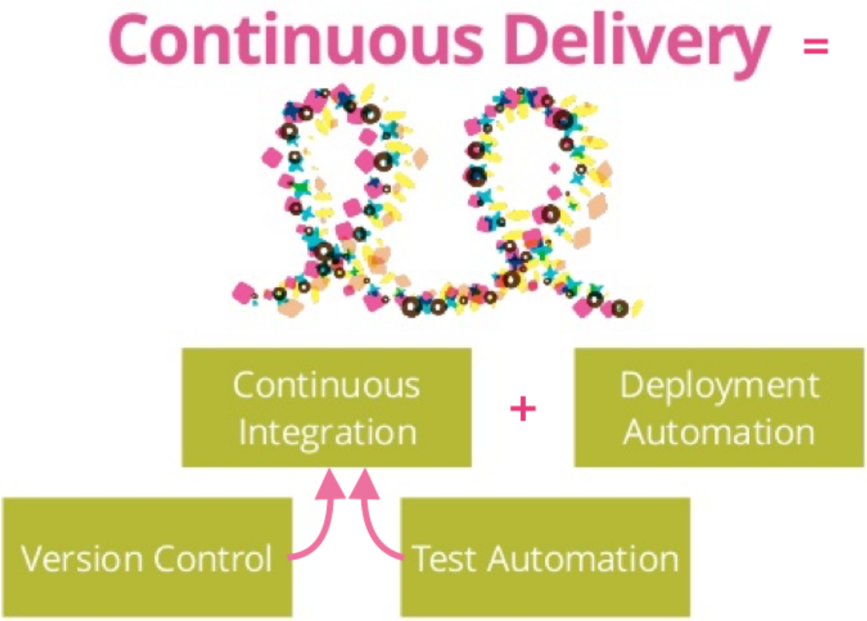
If you are using feature branches, if you have big and infrequent merges, or if the build run remains broken for hours, you are not doing it.

What good looks like

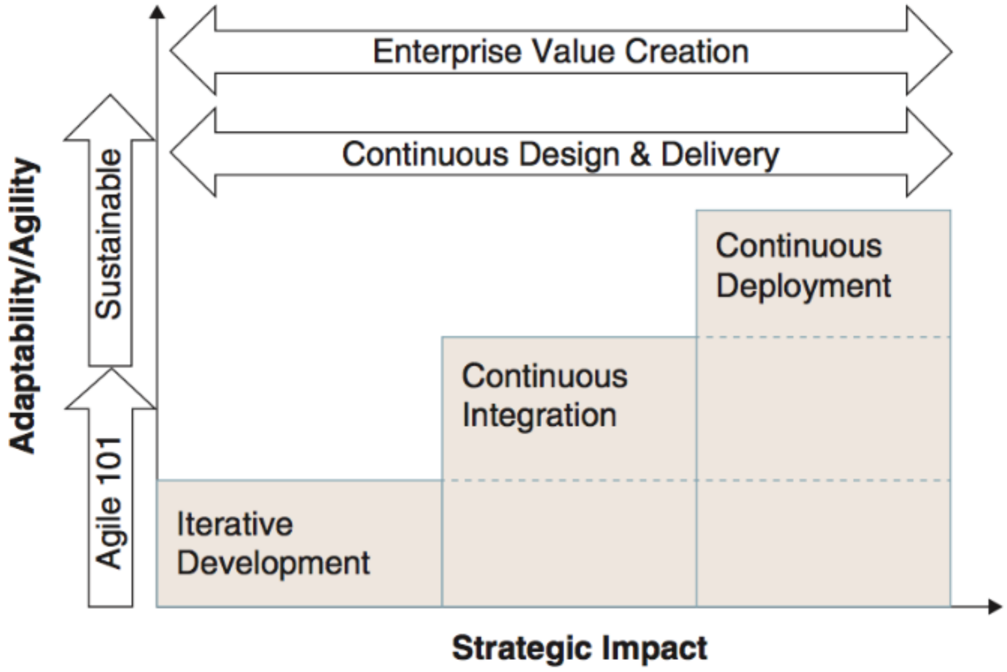
In the best organisations, integration is typically done many times per day, every 15 minutes or every hour. When the integration fails, either it is fixed within a few minutes or the changes that brought on the failure are rolled back.

Developers commit code on the mainline, practising trunk-based development and related techniques.

The following modified version of a ThoughtWorks image shows how continuous integration builds on top of other practices presented in this booklet.



The following image from Jim Highsmith's *Adaptive Leadership: Accelerating Enterprise Agility* shows how continuous integration builds on top of iterative development and how it enables other practices (continuous delivery and continuous deployment) that have strategic and business impact.



Next actions

Ask your tech leads to assess the status of your deployment automation, testing automation, and continuous integration and to identify the gap between what they have observed and what is described here in the “What good looks like” subsection. Ask them to identify actions to bridge the gap.

How we can help:

- We can assess your continuous-integration status, identify strengths and pain points, and recommend improvements and options for implementing them.
- Test-automation techniques are difficult to learn and master. They involve a steep learning curve that requires skills, maturity, and time, particularly when developers are entrenched in the “code then test” paradigm. In my experience, hands-on training is fundamental to climbing the learning curve and will let you avoid many costly mistakes.
- We can deliver the Agile Testing and QA Training session to help developers and testers to automate tests and practice test-driven development.

Do you want to know more on these subjects?

Here are some suggested readings:

- *Continuous Integration: Improving Software Quality and Reducing Risk* by Paul M. Duvall, Steve Matyas, and Andrew Glover
- *Agile Testing: A Practical Guide for Testers and Agile Teams* by Lisa Crispin and Janet Gregory
- *More Agile Testing: Learning Journeys for the Whole Team* by Janet Gregory and Lisa Crispin
- *Test-Driven Development: By Example* by Kent Beck
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce
- *Specification by Example: How Successful Teams Deliver the Right Software* by Gojko Adzic

Continuous delivery and infrastructure as code: Cutting-edge trends

This is the last section about practices in software development that CTOs should always have on their radar.

This section is dedicated to two trending practices in IT. As these practices are relatively new, you won't yet find scientific research on their effectiveness. Nor are these practices sufficiently broadly adopted in the industry to be considered fundamental building blocks of software development done right.

Instead, these two practices are adopted by top-notch digital and Internet giants and by some of the best digital companies.

Since these practices build on top of practices presented in previous sections, once a CTO manages to reach a good implementation of those building blocks (look at their "What good looks like" subsections) he/she should assess (explore with the goal of understanding the potential impact on the organisation of these practices) and run trials (understand how to build the capability to adopt those practices and experiment with them in circumstances where risk is acceptable) for these additional practices.

9) Continuous delivery and DevOps

Industry-trending practice

Companies such as Amazon, Google, Facebook, Flickr, Spotify, Etsy, SAP, and HP are adopting continuous delivery (CD) and DevOps.

The business impact of CD and DevOps, the growing number of nodes and complexity of production systems, and the virtualisation of production servers and the cloud are all driving the adoption of CD and DevOps.

Latest *State of DevOps Report 2017* from Puppet and DevOps Research and Assessment, is based on more than 27,000 survey responses in the past six years. That report classifies as low IT performers those companies whose deploy frequency is once per week and once per month or less frequent, and have a meant time to recover between one day and one week. While high IT performers deploys' into production are 46x more frequent, lead time for changes is 440x faster, and change failure rate is 5x lower.

Are you doing it?

According to the definition, you are doing CD when your software is deployable throughout its life cycle, starting from the first iteration/sprint and anytime during any iteration/sprint. Furthermore, your team prioritizes keeping the software deployable over working on new features, and anybody can get fast, automated feedback on the production readiness of their systems whenever somebody makes a change to them. Finally, you can perform push-button deployments of any version of the software to any environment on demand.

What good looks like

The two most common mistakes in CD and DevOps adoption are about continuous integration and DevOps itself.

Many organisations approach CD with a fundamental misunderstanding of continuous integration. They make heavy use of branches in the source-code repository and they don't practice trunk-based development.

The other fundamental misunderstanding is about DevOps. Many organisations create a DevOps team between developers and IT operations — but DevOps does not exist at all as role. It is really about a deeper collaboration between developers and IT operations in which some IT people move into developer teams and some developers rotate into IT-operations teams.

Releases into production are predictable, safe, regular events to the point that they become non-events. When deadlines approach, people are relaxed.

The quality of code released into production is good and IT operations are not in constant fire-fighting mode. Even when showstopper bugs appear in production, automated remediation plans make problems solvable in a few minutes without affecting users or the business.

Next actions

- Ask your tech leads to review the continuous-integration implementation and the trunk-based development practices and to identify gaps and plan actions to remove them.
- Ask HR to review the DevOps strategy and identify ways to remove the DevOps role in favour of a rotation between developers and IT operations.

How we can help:

- We can deliver the presentation “Continuous Delivery Overview: From Continuous Integration to Continuous Delivery and DevOps”.
  We can assess your CD status, identify strengths and pain points, and recommend improvements and options for implementing them.
  We can deliver CD training to bring developers, IT operations, and tech leaders to a better understanding of CD and help them learn the basics to drive a CD implementation.
  We can coach, mentor, and train developers and help them improve the continuous

integration, achieve trunk-based development, improve the architecture and the design of the product code base to make it testable, configurable, hot-deployable, and compatible with remediation plans.

Do you want to know more on these subjects?

Here are some suggested readings:

- *Continuous Delivery Overview* by Luca Minudel
- *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* by Jez Humble and David Farley

10) Infrastructure as code

Industry-trending practice

Infrastructure as code is a practice of CD. The growth of virtualisation and the cloud together with the possibility of dynamic resource allocation on the cloud make this practice especially important.

Do you want to know more on these subjects? Suggested reading:

- *Infrastructure as Code: Managing Servers in the Cloud* by Kief Morris

Appendix 1: Good decision making and 10x variations in productivity (references)

Augustine, N.R. 1979. “Augustine’s Laws and Major System Development Programs.” Defense Systems Management Review: 50–76.

Boehm, Barry W., T.E. Gray, and T. Seewaldt. 1984. “Prototyping Versus Specifying: A Multiproject Experiment.” IEEE Transactions on Software Engineering 10(3): 290– 303.

Boehm, Barry W., and Philip N. Papaccio. 1988. “Understanding and Controlling Software Costs.” IEEE Transactions on Software Engineering 14(10): 1462–1477.

Boehm, Barry, et al. 2000. *Software Cost Estimation with Cocomo II.* Addison-Wesley.

Card, David N. 1987. “A Software Technology Evaluation Program.” Information and Software Technology 29(6): 291–300.

Curtis, Bill. 1981. “Substantiating Programmer Variability.” Proceedings of the IEEE 69(7): 846.

Curtis, Bill, et al. 1986. “Software Psychology: The Need for an Interdisciplinary Program.” Proceedings of the IEEE 74(8): 1092–1106.

Cusumano, Michael, and Richard W. Selby. 1995. *Microsoft Secrets.* The Free Press.

DeMarco, Tom, and Timothy Lister. 1985. “Programmer Performance and the Effects of the Workplace.” Proceedings of the 8th International Conference on Software Engineering: 268–272.

DeMarco, Tom, and Timothy Lister. 1999. *Peopleware: Productive Projects and Teams* (Second Edition). Dorset House.

Mills, Harlan D. 1983. *Software Productivity.* Little, Brown.

Sackman, H., W.J. Erikson, and E.E. Grant. 1968. “Exploratory Experimental Studies Comparing Online and Offline Programming Performance.” Communications of the ACM 11(1): 3–11.

Schlender, Brenton. 1989. “How to Break the Software Logjam.” Fortune, September 25.

Valett, J., and F.E. McGarry. 1989. “A Summary of Software Measurement Experiences in the Software Engineering Laboratory.” Journal of Systems and Software 9(2): 137–148.

Weinberg, Gerald M., and Edward L. Schulman. 1974. “Goals and Performance in Computer Programming.” Human Factors 16(1): 70–77.

Appendix 2: Software design, modularisation and design patterns (references)

Oram, Andy and Greg Wilson, eds. 2010. *Making Software: What Really Works and Why We Believe It.* O'Reilly Media.

Prechelt, Lutz, et al. 2001. “A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions.” IEEE Transactions on Software Engineering 27(12): 1134–1144.

Prechelt, Lutz, et al. “Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance.” IEEE Transactions on Software Engineering 28(6): 595–606.

Unger, Barbara, Walter F. Tichy. 2000. “Do Design Patterns Improve Communication? An Experiment with Pair Design.” Workshop on Empirical Studies of Software Maintenance. http://www.ipd.uni-karlsruhe.de/Tichy/publications.php?id=149.

Vokac, Marek, et al. 2004. “A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns — A Replication in a Real Programming Environment.” Empirical Software Engineering 9: 149–195.

Appendix 3: Organisations, team structure, and Conway's corollary (references)

Bird, C., et al. 2008. "Latent Social Structure in Open Source Projects." SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT Symposium on Foundations of Software Engineering: 24–35.

Cataldo, M., et al. 2006. "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools." Proceedings of the 20th Conference on Computer Supported Cooperative Work: 353–362.

Nagappan, N., B. Murphy, and V. Basili. 2008. "The influence of organizational structure on software quality: An empirical case study." Proceedings of the 30th International Conference on Software Engineering: 521–530.

About the author

Luca Minudel is a Lean-Agile Coach & Trainer with 15 years of experience in Lean/Agile and 20+ in professional software delivery.

He is passionate about agility, lean, complexity science, and co-creation.

He contributed to the adoption of lean and agile practices by Ferrari's F1 racing team. For ThoughtWorks he delivered training, coaching, assessments and organisational transformations in top-tier organisations in Europe and the United States. He worked as Head of Agility in 4Finance and worked as Lean/Agile Practice Lead, and Lean/Agile Coach in the banking sector.

Luca is founder and CEO at SmHarter.com, the company that helps organisations turn their way of working into their competitive advantage.