cegeka

Agile Software

How to build in quality from the start

In a world where everything is enhanced by software, clean code is one of the fastest ways we provide business value to our customers. We create modular software that is easy to build on and delivers value quickly.

Of course, clean code does not write itself. It requires hard work by skilled people. We have over 700 people in our company and over the last 10 years we have adopted and adapted many agile software engineering best practices. Following the lean principle of reducing waste, we build quality into the development process. Figure 1 shows end users and the development team working together and applying a number of best practices.In this paper we explain these best practices, which all contribute to producing clean code, i.e. optimal code quality. We share best practices for both programming and testing. In theory, they are all more or less equally important for building high-quality future-proof soft-ware. But in practice, we choose which best practices to use according to the context and needs of each project.

Happy reading.

Quality assurance best practices

Quality assurance practices are designed to prevent quality issues from coming up at all. These practices, which can be split into programming practices and testing practices, are listed in Figure 1.

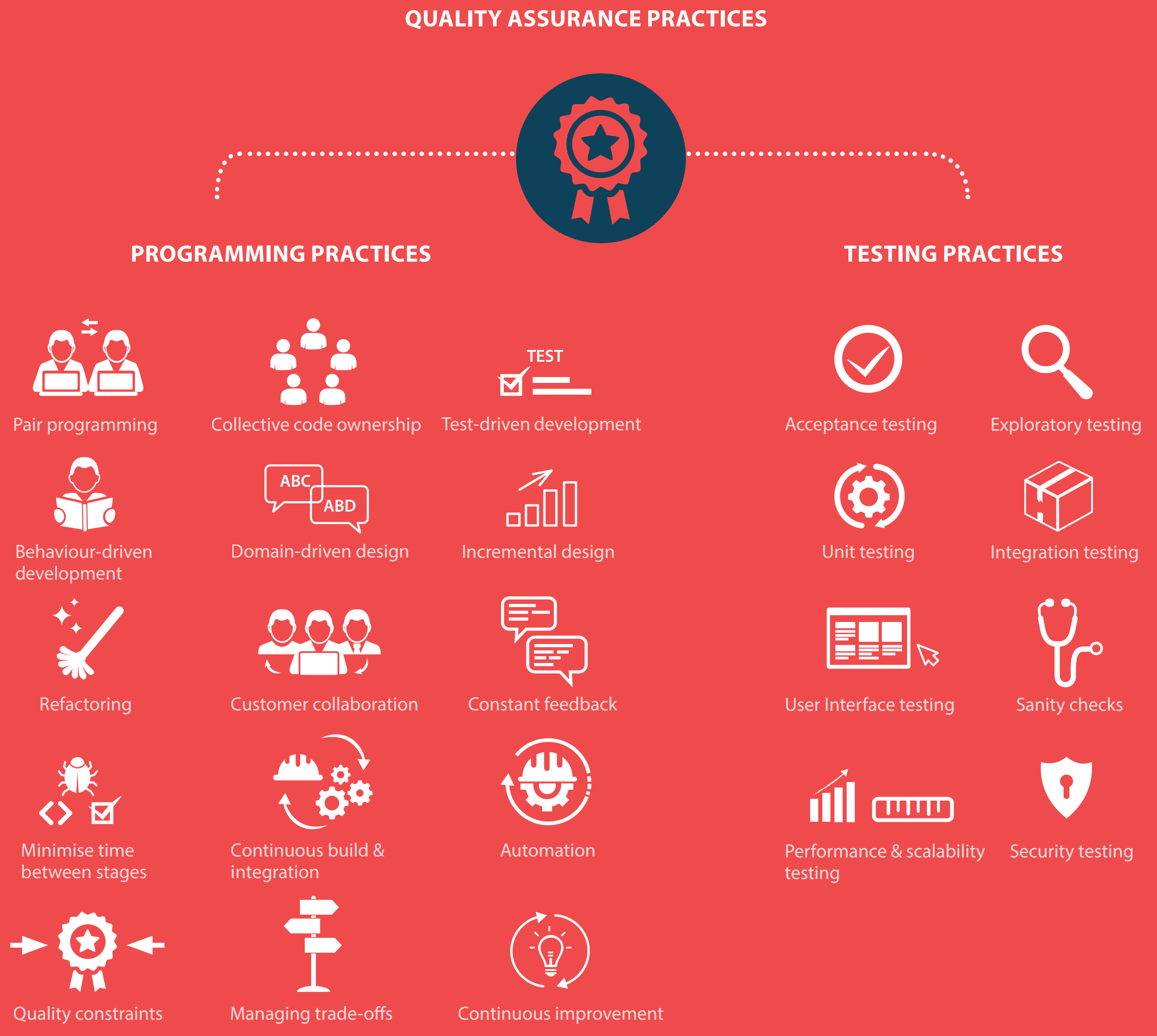


Figure 1. Mind map of quality assurance

CONTENTS

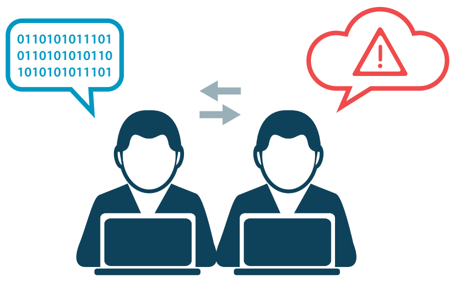
1. Programming practices

Pair Programming

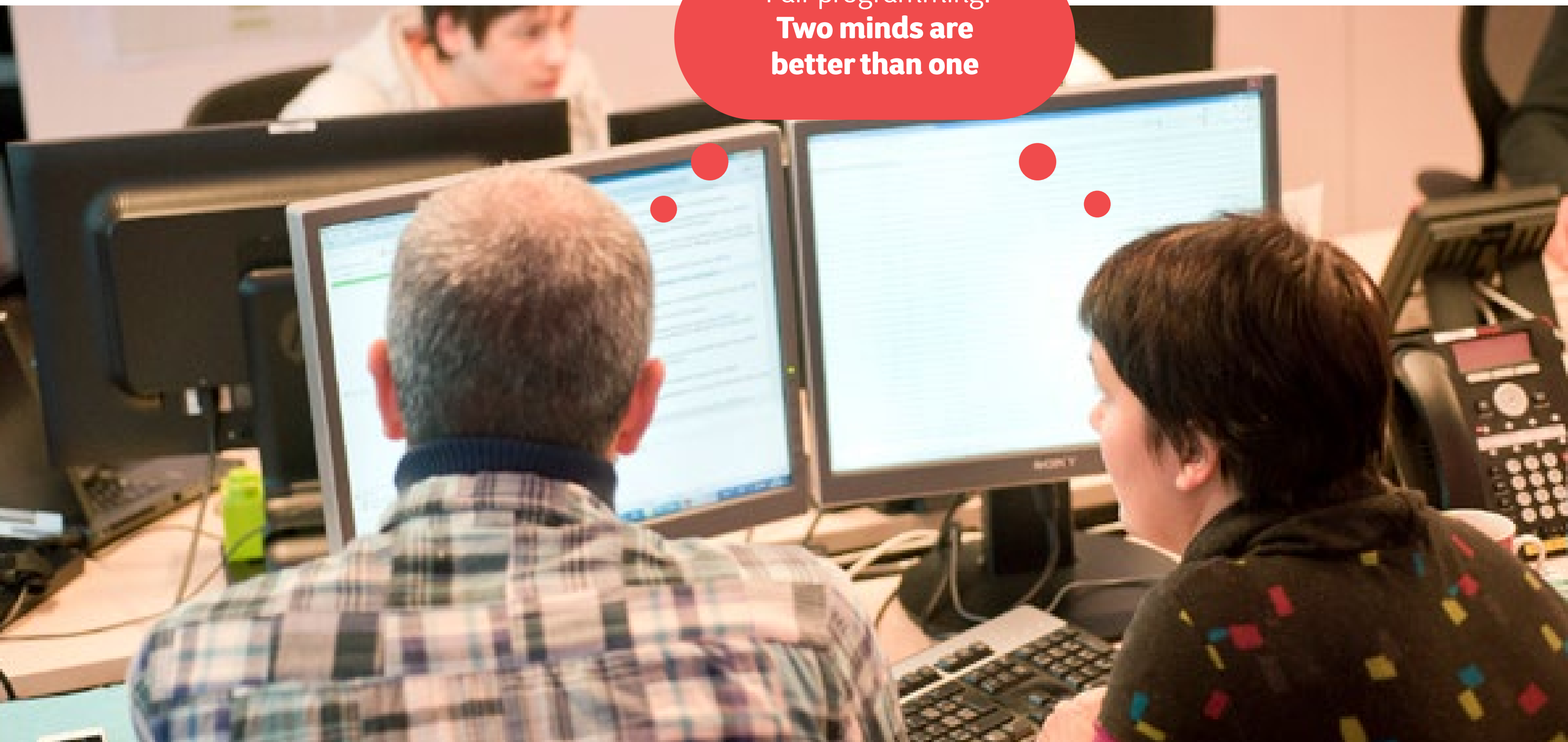
This component of Extreme Programming aims to avoid quality issues by having two developers working together simultaneously on a task, **one in the role of 'driver' and the other as the 'navigator'.** The driver types the code while explaining what he or she is doing and why. The navigator thinks ahead to the next steps and potential pitfalls, anticipating issues before they occur. The result is better quality code. The driver and navigator switch roles every 15 minutes, so this method is also known as ping pong programming.

By combining their experience, the developers often come up with solutions that might not have occurred to them alone. This often leads to better productivity. Additionally, this technique **guarantees knowledge transfer** as best practices and coding standards are shared more widely among the team. Not only does this result in more readable code, but it also contributes positively to employee satisfaction.

The team decides when to work in pairs. If code is developed by just one person, a code review is obligatory before a user story can be considered complete.



Pair programming:
Two minds are better than one



COLLECTIVE CODE OWNERSHIP

This practice means that any developer can edit any piece of code. So no one owns the code in an agile project – it belongs to the whole team. Anyone can start working on the next user story of the iteration. In addition, people are encouraged and expected to make any changes in the code needed to complete the work that the team is doing. Readable code is key to collective code ownership. So developers have to write **code that is easily understood** by colleagues on the team and will be self-explanatory to those maintaining the code in the future.

Collective code ownership is easier when the team follows **a consistent approach** to design and coding standards. So the team needs to establish a house style that everyone agrees to follow. A consistent style will ensure that their code is more readable and that less time is wasted on reformatting code to personal taste. Moreover, having a stable team with few staff changes will contribute positively to this practice.

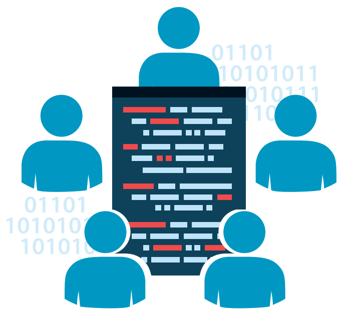
In this environment, there are **no 'heroes'** – individuals who position themselves as critical resources by being the only ones who know the code, and then have to come in at weekends or during holidays if problems occur. By implementing their quick-fix solutions, however, they can end up creating new problems for themselves. Collective code ownership makes this unnecessary.

TEST-DRIVEN DEVELOPMENT (TDD)

This component of Extreme Programming aims to avoid quality issues by **writing tests before writing code**. Basically, the analyst writes down the test conditions for each feature just before it is developed. If the developers know how the code is going to be tested, they are much more likely to write code that meets all the requirements.

In our Software Factory, it is a best practice to write **automated unit tests** for each of the test conditions before actually writing the code. The developer then writes the code needed to pass the tests. TDD is also useful for regression testing after software maintenance, because it simplifies the process of checking whether changes to the code during maintenance have any negative effect on the software.

Since TDD means building only what is needed and testing that it works, it leads to a much **easier code base to maintain and modify**. With less code comes less complexity, and this simpler design means that changes and modifications become much easier. Another positive result is that the documentation is an inherent part of the code, so it is always up to date.



BEHAVIOUR-DRIVEN DEVELOPMENT (BDD)

Behaviour-driven development means that **customer specifications** are used as input for the development. The specifications are written **in the language of the customer** and describe the behaviour the customer expects from the application. While developing the functionality, the developers will turn these specifications into automated tests, by linking the test in the language of the customer to the code, using tools like JBehave.

The difference between the unit tests (written using test-driven development) and automated integration tests and these behaviour-driven development tests, is that the first two focus on the code, while behaviour-driven development tests **focus on the behaviour the customer expects** from the code. Through regular execution of these tests, building them in in the continuous integration and deployment process, we now cannot only assess that the code is clean and unbroken (unit and integration tests), but also that it does what the customer expects it to do. Since these tests are written in the language of the customer they also offer a living documentation of the intended behavior of the system, that is constantly tested.

USER STORY

User stories describe in one sentence, in the customer's terminology, a small piece of functionality used to incrementally build the application. It represents business value, as it captures what a user does or needs to do as part of his/her job.

The user story is not a form of documenting business requirements, but more a way to promote having a discussion amongst team members who will actually work on the story, together with the business, who needs the outcome of the story.

It is essential that the user story precisely defines the objective of the code and has the following characteristics:

Independent

it is important that teams can develop, test and deliver the user story on its own and that it can be valued independently.

Negotiable

the user story is not a contract for features, but rather a placeholder for requirements to be discussed, developed, tested and accepted, negotiable between the business and the team.

Valuable

each story represents business value and provides end-to-end functionality (slicing through all layers of the architecture).

Estimable

each story contains enough information to be estimated by the developers. If the story is too uncertain to estimate, a spike can be used to reduce the uncertainty and produce user stories that are estimable.

Small

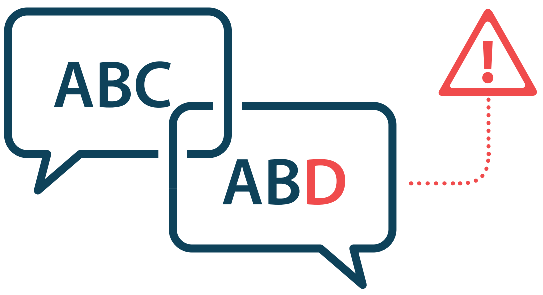
a user story can be implemented in 1 iteration.

Testable

acceptance criteria and a test design are agreed for the user story to minimise interpretation problems and to use for confirmation that the software is working.

DOMAIN-DRIVEN DESIGN (DDD)

This is an approach to software development for complex requirements that connects the implementation to a model of the domain the software will be used in, using a language that is shared by the team and the customer/users. **If the words used in the software do not precisely match those used by the business, it can lead to all kinds of problems.** In agile development, it is the developer's job to speak the language of the user, not the user's job to speak the technical language.



INCREMENTAL DESIGN

This simply involves taking time to improve the design of the software in small steps as you go along. Design improvement becomes part of every developer's day, so that it is applied to every user story and not left for later. By working in this way, developers think about the design of the software as they write tests, as they implement the code to pass these tests, and before they check in their code. When developers create a design for their current user story, they bear in mind upcoming user stories in their design decisions. In addition, reworking a design often improves it. Each time the design is reworked, it becomes more refined and malleable.

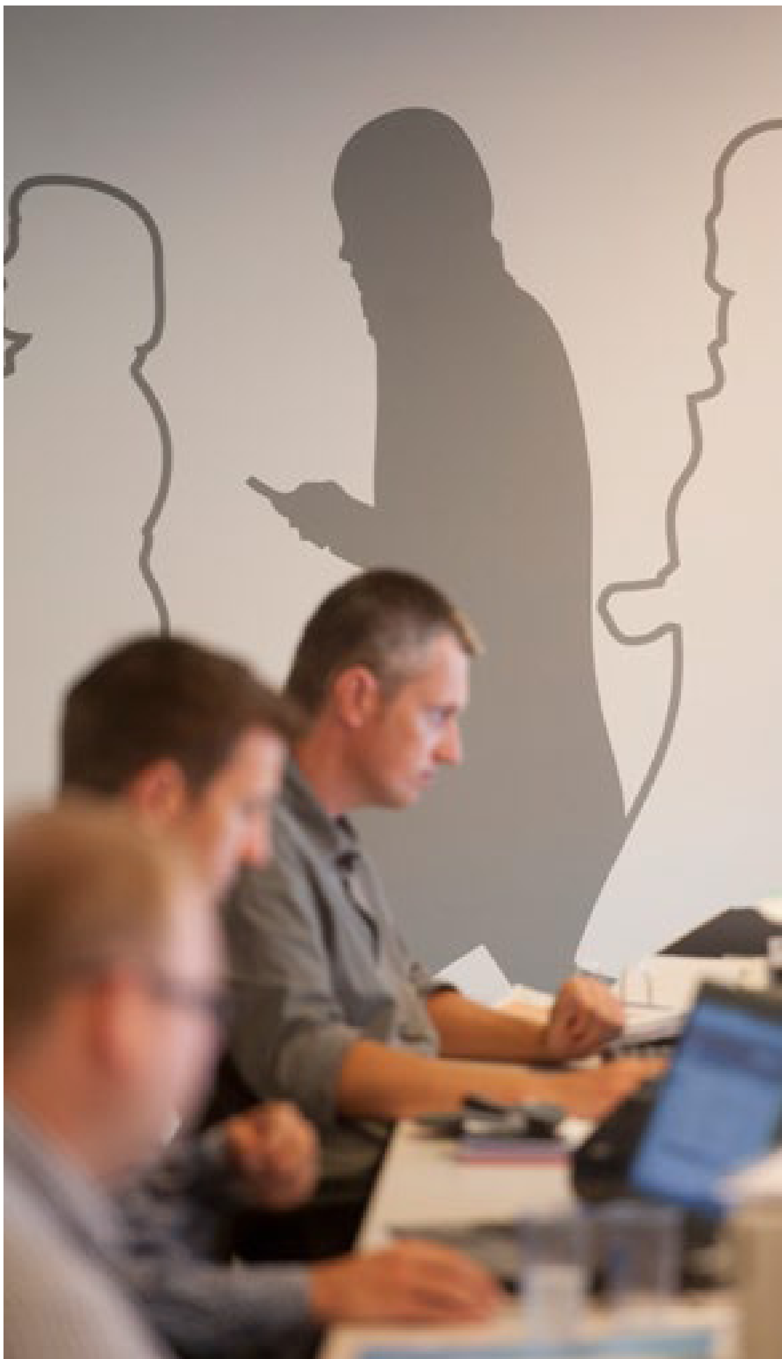
Making the shift from standard design to incremental design right from the start can be a big challenge. In practice, limited time is spent on thinking about design without producing working software. Developers prove their ideas by implementing them, designing for current needs and keeping their design as simple as possible.

REFACTORING

This is a method to **systematically improve the structure of the code without impacting its external behaviour**. It involves placing the code in the right place where other developers can find it quickly and easily, and keeping the code organised and decluttered. Refactoring also means using well-named methods and variables that improve the readability and maintainability of the code base. This is crucial for **reducing 'technical debt'**. Comparable to real-life debt, technical debt often has to be paid to rectify faults from the past in order to make progress again (e.g. code that was badly designed). However, if the debt is not controlled, it can become one of the project's biggest risks and significantly reduce productivity. You can compare it to a car mechanic who never cleans the floor after spilling oil. If this goes on for years, they can try scrubbing the floor as much as they like, but they will never get the floor clean. It would have been better to have given the floor a quick wipe every day.

Refactoring does not involve the aesthetic organisation of the code; it is basic housekeeping. Importantly, refactoring should not affect the proper functioning of the code. As a result, refactoring requires excellent test coverage, and it is inextricably linked to test-driven development. Refactoring is done continuously throughout the day, with small steps and improvements. The team is allowed to refactor anywhere in the code base: developers rename methods and variables to make the code base easier to read and understand.

Major refactoring is rare and addressed in a separate technical story that, like any other user story, needs to be estimated and prioritised. Aggressive refactoring near the end of a project does not slow you down, it speeds you up. Adding new code and fixing bugs results in better, faster and cheaper software in the long run. It keeps the change capacity of the application high, which means that even older applications can be kept operational without exposing the business to risk.



CUSTOMER COLLABORATION

Close involvement of the customer – the business side as well as the IT side – creates the **dialogue needed to define requirements correctly and to clear up possible misunderstandings immediately**. A collaborative team also verifies that the implementation is correct through the use of exploratory and acceptance testing. In this process, it is important to focus on the features that business needs most, based on its vision, with the functionality reflecting the biggest value for the users. If users see the added value of the software, it usually minimises the functional assistance they require.

Customer collaboration requires in-depth knowledge of the process, with a customer committed to the project. The customer's validation team should visit the development team frequently, especially during testing. This will give them the opportunity to explain how the customer's needs fit into the bigger picture and also lead to better informed decisions by the development teams. It is a good practice for the validation team to present this information visually, using information panels, posters, whiteboards, etc.

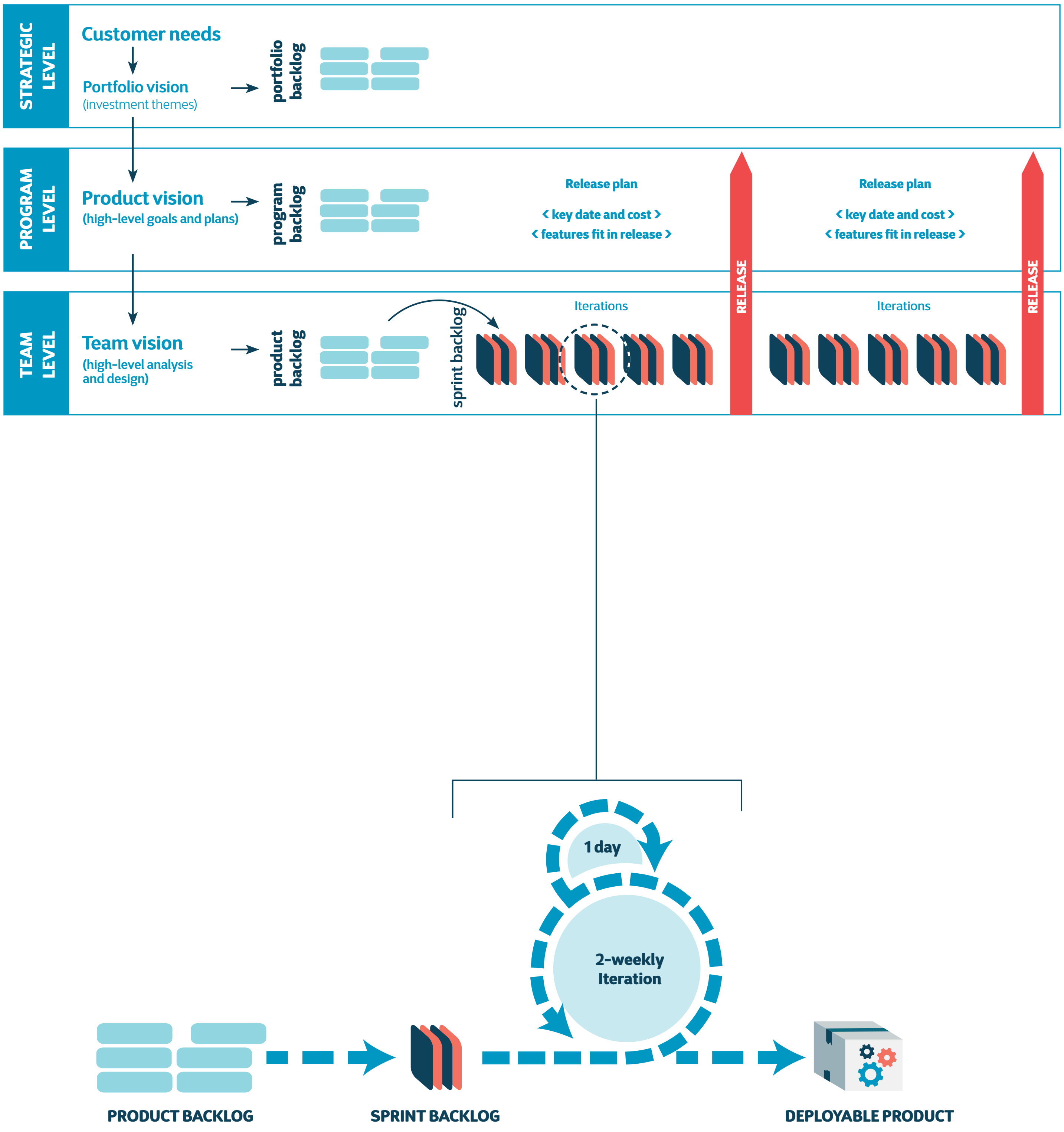


Figure 2. Customer collaboration – Deep involvement of the customer ensures definition of the right requirements

CONSTANT FEEDBACK

Both Scrum and Extreme Programming build quality into the process by developing in small incremental steps and using short iterations. These agile methods allow **the customer and the team to work together closely** with constant two-way feedback between them. This feedback can considerably enhance the quality of the code, because it means the developers inspect and adapt the product every single day to ensure the right level of quality, and more importantly, the right product.

Feedback also makes it possible **to set priorities,** if necessary by comparing the cost consequences of a reported bug with the cost of fixing it. The practices of Extreme Programming and Scrum are completely complementary, so development teams can choose to use both.

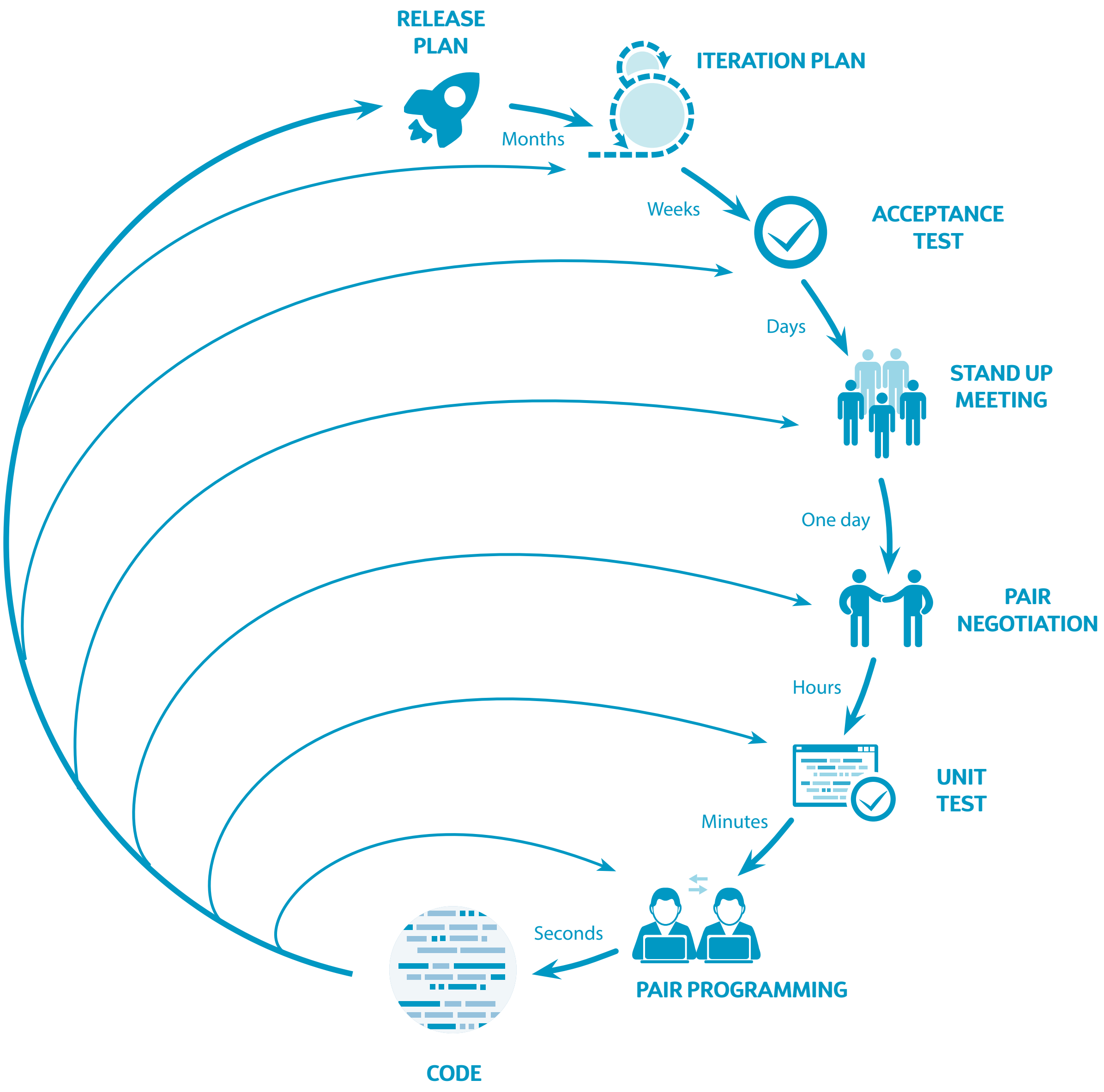


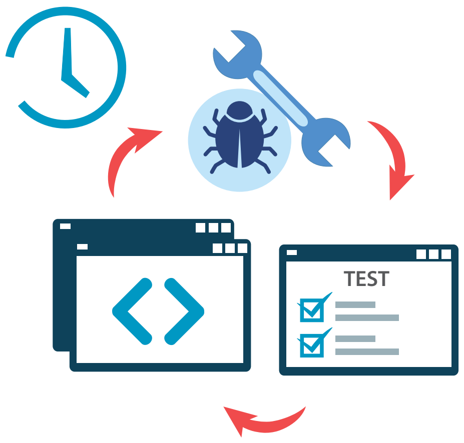
Figure 3. Planning/feedback loops: constant feedback at different levels helps to build in quality.

MINIMISE TIME BETWEEN STAGES

This important technique for building quality into the development process **reduces the time between development, testing and bug fixing to a minimum.** Instead of logging bugs, developers resolve them immediately. In most cases, logging bugs is a waste of time. Testing the code as soon as it is developed and fixing bugs as soon as they are found eliminate the need to log them. Moreover, a long period between writing the code, testing it and fixing bugs disrupts continuity and results in delays due to task switching, knowledge gaps and a lack of focus.

A useful tool here is our **'definition of done'** – a list of requirements that should be met before a user story is considered complete – which allows the team to do a final check independently of each other.

There is a positive correlation between high speed for resolving defects and the quality of the software.



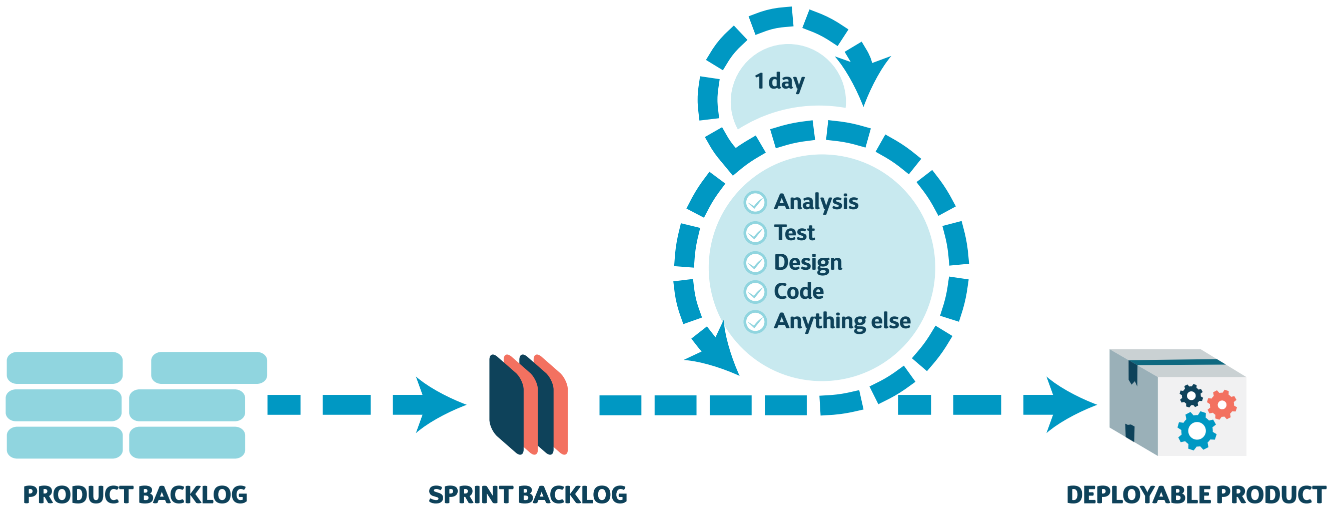


Figure 4. Perfect timing: minimising the time between development, testing and bug fixing results in better quality software.

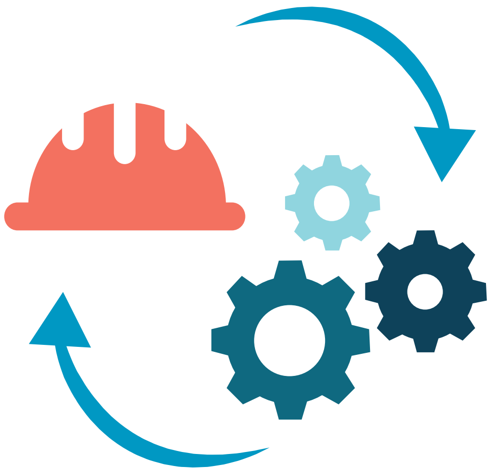
CONTINUOUS BUILD AND INTEGRATION

Most agile methods recommend doing regular and frequent builds, at least daily if not hourly: “Integrate often and fail fast!” Extreme Programming recommends continuous integration, with **code built and automatically unit tested** as soon as it is checked-in and then integrated into the overall system. Reducing the gap between builds to a minimum also cuts down the time spent on integration. On major waterfall projects, the integration and regression testing phases can be very long. This problem can be avoided by regular builds and frequent integration.

We use **build scripts** that integrate the new code with the existing one to form a working whole. This happens before the code is committed to the version control system, so that problems (in the new code itself, or between the new code and external components) are quickly identified and resolved. The teams use a source code repository and behave as if they are in production from day one of the project.

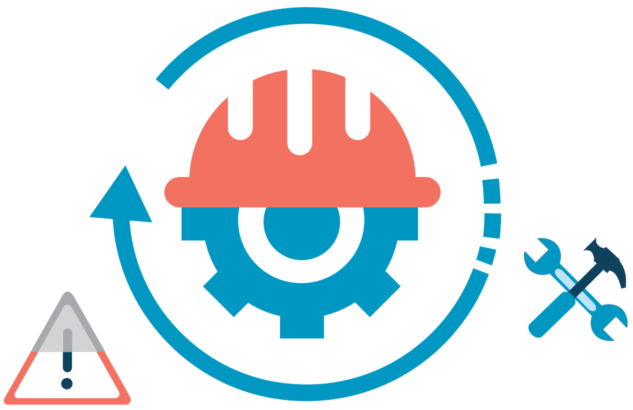
Developers establish a check-in process and integrate code in small chunks multiple times a day. Continuously integrating the changes committed by developers throughout the day can make the whole process of building, integrating and deploying software fast and easy. Not only does continuous integration of code squash bugs earlier, but it also cuts the cost of changes and leads to confident deployment.

Moreover, the more automation, the better.



AUTOMATION

The teams use an automated build system and developers always **fix a broken build immediately**. In addition, automated regression testing can be used to reduce the work involved in detecting quality issues before they occur in a live environment. Where applicable, all repetitive tasks should be automated as much as possible, so reducing the risk of human error.



QUALITY CONSTRAINTS

Non-functional requirements are also described in the form of user stories if possible, for example in the field of usability ("As a user, I want to be able to navigate through a form with the tab key"), performance ("As a user, I want an answer within x seconds of submitting a form"), scalability ("As a system, I can support 100 concurrent users without impacting response times"), etc. Such requirements are also called quality constraints, because they define the limits of the functionality implemented.

An important rule is that constraints should be **incorporated after a user story is completed in an iteration**. For example, consider the requirement about submitting a form and wanting an answer in 0.2 seconds. The performance of the form may not be a priority during the initial iterations, but after running this quality constraint all subsequent forms should react within 0.2 seconds. The mechanisms necessary to validate this performance should also be established (e.g. automated performance tests).

Some quality attributes are so obvious and inherent in our software engineering practices and ways of working that we want to respect them at all times. Examples include test coverage, clean code and design practices. These quality constraints are respected from the very first user story onwards and form an integral part of our definition of done: the **constraints have to be met before a user story is considered complete**. Certain architectural agreements made with the customer can also be treated in this way, for example naming conventions, service orientation, etc.

MANAGING TRADE-OFFS

Bear in mind that **quality** is only one aspect of a project – **time, cost and scope** should also be taken into account. Sometimes there are commercial reasons to trade off quality against other factors.

There may also be **situations where focusing on quality costs more than the issues the developers want to avoid.** An example of agile methods recognising a trade-off in theory is the acceptance of reworking (refactoring) due to not having detailed specifications and a complete design right at the start of a project. In traditional methodologies, detailed functional and technical analyses were designed to improve quality early in the project life cycle. But over the years, many people discovered they were counterproductive – and so agile methods were born.

Likewise, if developers are building fairly low-complexity visual features that have a limited impact, it may be better to spend less time on quality assurance, because the risk of quality issues coming up is low and even if they did, their effect would be small. Of course, this is a judgement call and it may be difficult to know where to draw the line.

In order to **work on the right features at the right time**, the team should prioritise the product backlog. This will help them to focus on the features that the business needs the most. The best approach is for the team to work according to a fixed time and materials budget, as this will give them the flexibility to react smoothly to any priorities set by the customer.

When setting priorities, you can minimise risk by planning to do complex user stories with technical uncertainty at the beginning of a project.

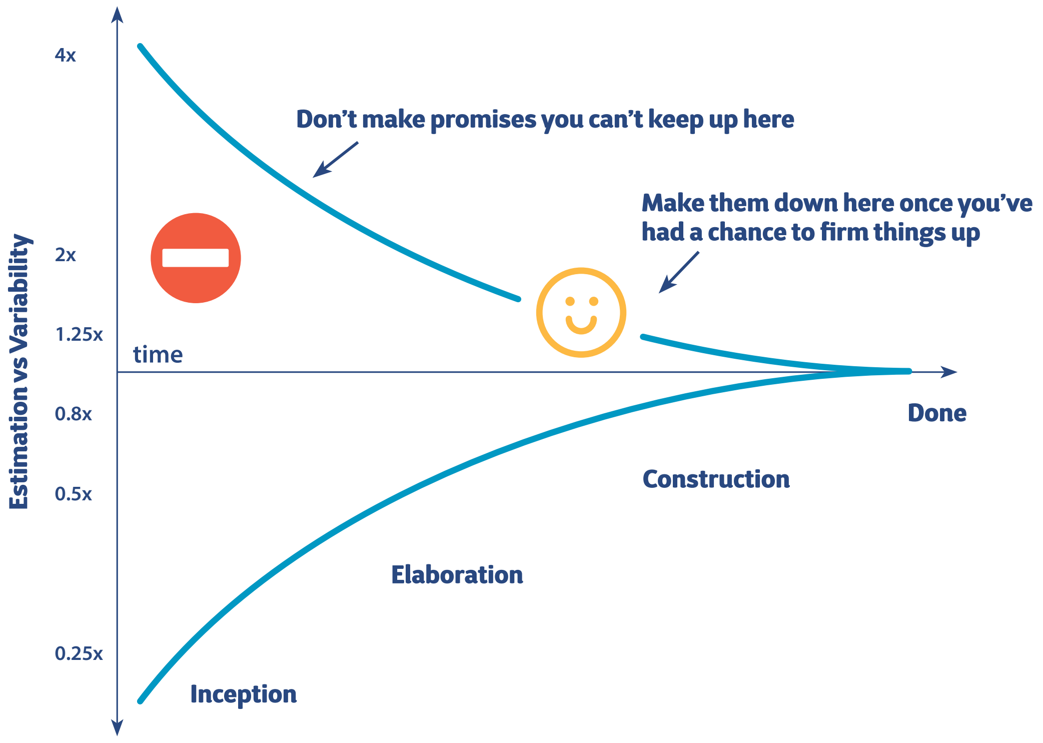

Estimation vs Variability
4x
2x
1.25x
0.8x
0.5x
0.25x
time
Don't make promises you can't keep up here
Make them down here once you've had a chance to firm things up
Done
Construction
Elaboration
Inception

Figure 5. Reality check: the accuracy of estimates can depend on the stage of the project.

CONTINUOUS IMPROVEMENT

Retrospectives (feedback, inspect and adapt) – after the sprint, demos, code reviews, extensive testing, mature project management methodology, active coaching and so on – **help people to continuously improve** their ways of working, within and across the teams. Apart from specific projects, we offer a number of ways to help our people gain new insights and to expand their knowledge, including internal knowledge sessions, regular training, reading groups, competence centres, etc.

2. Testing practices

Besides quality assurance practices, we perform a number of systematic tests to detect possible bugs at a very early stage of development.

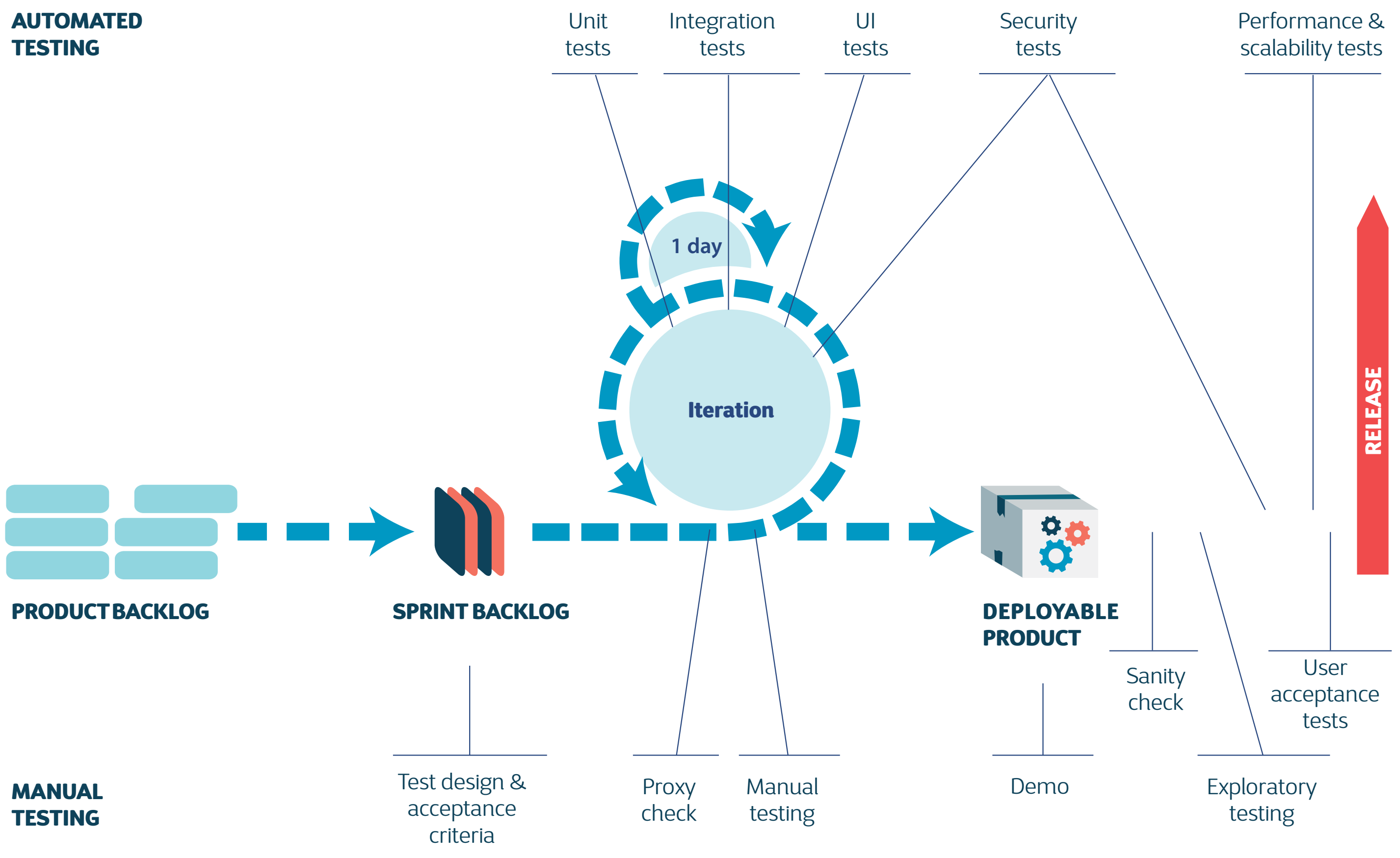


Figure 6. Reality check: the accuracy of estimates can depend on the stage of the project.

ACCEPTANCE TESTING

Each user story is based on a number of **objective criteria** that can be used to check whether the functionality has been implemented correctly. The customer defines the acceptance criteria and they are documented by customer proxies and added to the user story in a concise way.

Developers use the acceptance criteria to build the product, and customer proxies and the customer use them during the final acceptance tests. Some of the acceptance tests can even be automated, so that the acceptance criteria can be checked continuously when changing the software. This automation can be done using tools that allow test scenarios to be written in the language of business users and executed via a wiki. The customer proxies and developers work together to automate the scenarios. The set of acceptance test scenarios based on a set of user stories (features) is called a system test.

EXPLORATORY TESTING

Exploratory testing is carried out without predetermined test scenarios. It is **performed by the customer** after each iteration. The tester gradually discovers how the application works by trying different inputs and navigation paths. This way of testing resembles the way an end user will interact with the application. Exploratory testing often reveals more relevant issues than strictly adhering to defined test scenarios.

UNIT TESTING

This is an **automated way to test separate functional modules** or units. Unit tests slice through the entire application, testing everything from the application's business logic down to the database. These tests are basically independent of the rest of the code. If a unit is dependent on another unit, a mock framework is used to enable proxy implementation of certain objects.

There are many benefits of writing unit tests for the code:

- It gives instant feedback when changes are made to the code and a unit test breaks.
- It greatly reduces debugging time, because if a unit test fails, developers know exactly where the problem lies.
- The cost of regression testing is significantly lower, because retesting everything manually is not necessary when releasing a new part of the code.
- There is greater confidence when deploying to production, because there is a suite of automated tests validating the code.

INTEGRATION TESTING

Integration tests **are automated tests that focus on a certain group of logically linked components.** There are two types of integration tests: black box and white box. Black box integration tests only use the interfaces of the components to test the functionality. White box integration tests are performed in combination with another unit or component that is closely linked, for example a repository and its database, or a gateway and its external system. The white box integration test can directly contact the database or the external system for performing the necessary validations.

USER INTERFACE TESTING

User interface tests navigate through the **graphical user interface** (GUI) of the application to validate the screen flow and content (not the business logic or layout). In the case of a web-based application, these tests can be automated using tools like Selenium.

SANITY CHECKS

Sanity checks are tests that are performed **after each new installation of the software**, for every environment in which the software is installed. They are used to test the basic operation of the integrated system (e.g. application launch, access to external interfaces, etc.). Only when the checks are successful is it worthwhile performing other tests. Sanity checks should provide instant feedback to save time, so they should preferably be automated.

PERFORMANCE AND SCALABILITY TESTING

Performance and scalability tests are **two types of non-functional tests.** Performance tests verify the efficient operation of the application, measuring response times of the GUI or service. They can be automated by providing timing information when executing integration or acceptance testing. Scalability tests examine how the performance evolves with an increasing number of concurrent users. Both tests require the availability of a dedicated environment that is representative of the subsequent production environment(s).

SECURITY TESTING

Security testing, like any other type of test, is built in in our agile process and **part of the daily work** of an agile team. It is included in the task list for each sprint to protect business from security threats without slowing down the release cycle.

While testing the functional part of a story, authentication and authorization is also tested. Those tests can be automated.

For more specialized security testing, a **security specialist** from the Cegeka Security Office helps the team in defining tests, setting up tooling and testing the security of an application.

3. Make it happen

What do you need in order to implement agile best practices?

It is all about people. This will be no secret to anyone who has ever been involved in software development. Garbage in means garbage out. So for teams to perform as well-oiled machines, they need the right people, with the right culture and tools to support them.

Here we share our experiences in:

- Getting the right people on board.
- Keeping the right people on board and helping them to develop their skills.

3.1. Our recruiting code to get the right people on board

Over the last 10 years, Cegeka has come a long way in refining its agile development practices. However, finding experienced agile developers, functional analysts and project managers for our Agile Software Factory is far from easy.

We cannot simply ‘hire people away’ from competitors. Instead, we need to find the people with the right attitude and coach them to become disciplined agile practitioners.

Job **variety** thanks to different projects in different sectors

Competence centres

Growth opportunities in technical and functional roles, or as an agile coach

Become a (better) agile practitioner

MAIN REASONS WHY PEOPLE JOIN THE CEGEKA AGILE SOFTWARE FACTORY

Continuous Learning Environment

Job autonomy and self-managing teams

Engagement at all levels, teamwork (also with people from our customers)

3.1.1 Evangelisation & experience

We still need to go out and do a lot of evangelisation, to tell the world about agile. Unlike 10 years ago, most people in software development know about agile project management and agile development. However, our experience tells us that although people may know the word 'agile', it does not mean that they can actually apply agile principles.

An attitude of learning

There are few people out there who are fully aligned with agile and Cegeka's best practices from day one. That is why we look for people with a real drive to learn – people who are capable of questioning the status quo and developing themselves and even the methods we use. Our recruiters and hiring managers look for people who will not just blindly follow this or that best practice, but have the capability to understand the 'why' of our ways of working.

From evangelism to experience

To find the right people, we evangelise at job fairs and events. We organise Software Factory visits for candidates so that they can experience the atmosphere, see our teams at work and talk to our people. Further along in the recruiting process, we invite candidates to spend a half or whole day paired with one of our team members. This allows candidates to experience for themselves what it is like working at Cegeka. After all, people work to satisfy both rational and emotional needs. At Cegeka, we have nothing to hide and it is a fully transparent workplace.

Peer-to-peer recruitment

We think it is important to tap into the social networks of our people, have them suggest good candidates and reward them if 'their candidate' is hired. Of course, our people get social recognition, but they also earn a modest referral bonus. After all, by making a referral they are putting their reputation on the line.

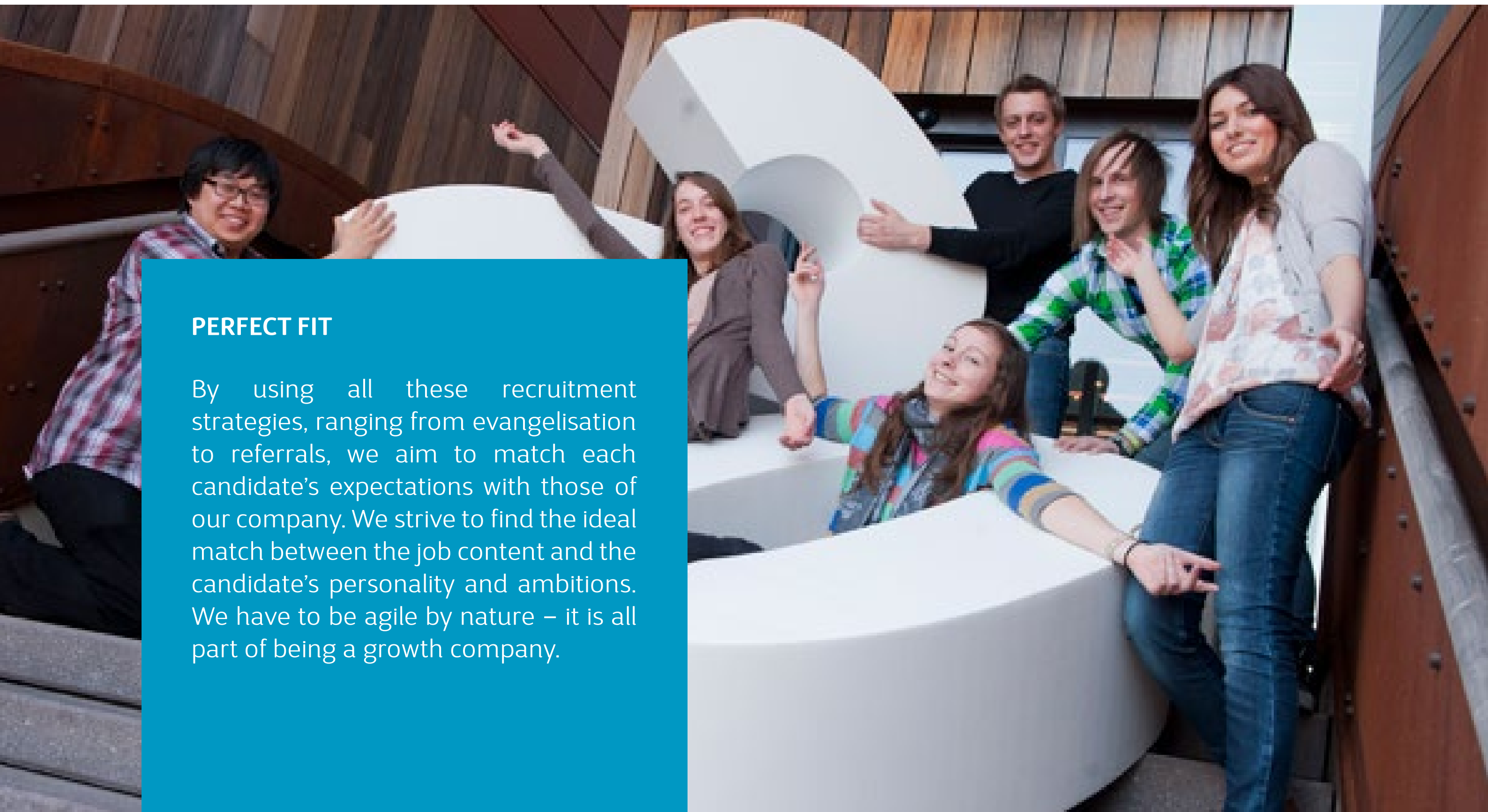
Social media

The war for talent has escalated to the digital level too, so we also use digital marketing techniques in our employer branding and recruiting activities. We target people with a certain job type or in a certain area via LinkedIn, or post our jobs on Twitter and Facebook via corporate accounts and spontaneous social sharing by our employees.

3.1.2 Graduate recruitment

Our recruiters and leadership also have good contacts at universities and colleges. They evangelise at job fairs and events as well as organise Software Factory visits, especially for students about to graduate.

Young graduate intake
We have a healthy intake of new graduates. So we can ensure a healthy mix of team members – senior, intermediate and junior profiles – and this also leads to a good market fit. Having different skill levels is useful as not every task requires a senior team member and it would be unfair to give a too junior team member the role of scrum master. If you do not manage your team members wisely, things will go wrong.



PERFECT FIT

By using all these recruitment strategies, ranging from evangelisation to referrals, we aim to match each candidate's expectations with those of our company. We strive to find the ideal match between the job content and the candidate's personality and ambitions. We have to be agile by nature – it is all part of being a growth company.

3.2. Our culture to keep the right people on board

With an employee turnover that is 20% lower than the sector average, we can keep the right people on board. We believe that our agile culture is the primary driver in retaining our people, closely followed by continuous learning (including training) and knowledge sharing.

Learning is part of agility
Learning is an inherent part of agile practices. Each new project is a learning experience in how to set up teams, how to discover the business value, how to work with the client organisation, etc.

Employees are part of self-organising teams that often work in pairs of junior and senior people. This helps to quickly raise the skill level of our junior employees. The teams also organise retrospectives to share learning points and identify improvements in the way they work.

Self-development
Employees take responsibility for their personal development. They can follow formal training programmes externally, join internal courses, become part of an innovation community, or participate in guilds. Guilds are people from different teams with the same function who come together to share their learning and best practices.

Twice a year, Cegeka organises a larger Knowledge Sharing Meeting where people from all divisions participate. The content of the meeting is developed bottom-up: people can suggest topics for a presentation or workshop that they would like to give.

A less formalised initiative are reading groups that share the learning points of books with new thinking about software development or agile project management. The idea behind this is simple: you can digest the content of one book a month, but you only have to read and summarise one a year.

Participation in learning initiatives

20% of the Software Factory employees attend **competence and innovation centres every year**

30% participate **in guilds**

80% participate in **reading groups**

60% attend **knowledge sharing meetings**

Looking over the fence

Employees are encouraged to participate in all kinds of external user groups for certain technologies (Java, .NET, etc.) or methodologies (e.g. the Agile Consortium).

Several times a year we also invite internationally recognised speakers in different domains to give presentations or workshops. We have opened these sessions to customers and third parties.

Organisation 2.0: shared responsibilities

These initiatives are on offer, but people are still responsible for their own growth and no one is forced to participate. However, learning and development are an integral part of performance appraisal discussions. So Cegeka is an example of an organisation 2.0: both parties, the employer and the employee, have a responsibility to make things work.

Cegeka's Agile Software Factory has been a leader in the field of agile software development and business application management for complex processes for over ten years. Cegeka currently has a team of over 700 developers. This allows us to work efficiently, and to offer you an approach that allows you to respond quickly to changes in or around your company.

Visit www.cegeka.com/en/ro for more information.

How can we help?
Let's discuss; come and visit our company.
E-mail: ro@cegeka.com

HEADQUARTERS:
Tower Center Boulevard
Ion Mihalache 15-17
Sector 1
011171 Bucharest
Romania

FOLLOW US ON

ro@cegeka.com
www.cegeka.com/en/ro
www.linkedin.com/company/cegeka
www.twitter.com/cegeka
www.facebook.com/CegekaRomania