Chapter 4
Quality Control

To counter quality decay during software evolution, proactive countermeasures need to be applied. This part of the book introduces the concept of continuous quality control and explains how it can be applied in practice. Particularly, it discusses the most relevant quality assurance techniques.

4.1 Quality Control Loop

The idea of continuous improvement processes as the basis for quality management systems originated in non-software disciplines in the 1980s [52, 114, 188]. It is now a standard approach for process-oriented quality standards such as the ISO 9001 [91]. A popular model for continuous improvement is the *PDCA cycle*, also called *Deming cycle* or *Shewhart cycle*. The abbreviation PDCA comes from the four phases of the model: Plan, Do, Check and Act. They are positioned in a loop as in Fig. 4.1.

In the *Plan* phase, we identify improvement potential, analyse the current state of the process and develop and plan the improvement. In the *Do* phase, we implement the process improvement in selected areas to test it. In the *Check* phase, we evaluate the results of the *Do* phase to be able introduce the improvement process in the *Act* phase. As you realise, the PDCA cycle is very general and can be used for software processes as well (see Sect. 4.6).

4.1.1 Product Quality Control

Not only processes are in need of continuous improvement, however. Various researchers have shown that software ages and undergoes a quality decay if no countermeasures are taken [16, 170]. At first sight, this insight is counter-intuitive,

S. Wagner, *Software Product Quality Control*, DOI 10.1007/978-3-642-38571-1_4,

© Springer-Verlag Berlin Heidelberg 2013

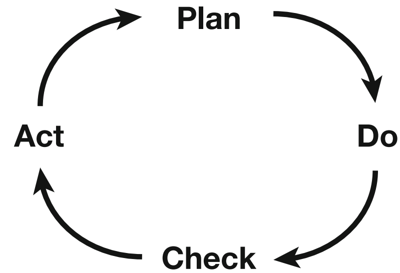


Fig. 4.1 The PDCA cycle

because software is intangible and, hence, not subject to physical wear and tear. A 10-year-old software does not work worse because its bits eroded over time. Nevertheless, an ageing process takes place also for software.

All successful software system change during their lifetime. This is commonly called *software evolution*, although this term is counter-intuitive as well. Software changes are not random mutations from which the best survive. Quite the contrary takes place: The software engineers perform maintenance to fix existing defects, to restructure the system, to adapt or port it to new or changed platforms and to introduce new features. This is not random but very deterministic. These many changes to the system are necessary so that the system stays valuable for its users, but they are also a danger to the quality of the system.

In practice, owing to time and cost pressure but also missing training, changes are done too often carelessly of the effects on the quality of the system. For example, developers implement calls to other components that were not allowed in the original software architecture to get the system running on a new operating system version. New features that are similar to existing ones are implemented by copying the source code for the existing systems and making changes to the copies instead of introducing common abstractions. Algorithms are introduced because they were the first to come to mind not because they are the most efficient. Wild hacks are introduced to make changes work because nobody understands the original code anymore.

Hence, this affects all quality factors: buggy changes reduce reliability and bad algorithms reduce performance efficiency. The most stroke quality factor, however, is maintainability. The source code and other artefacts become more and more incomprehensible and a nightmare to change. At the same time, this is far less visible to management, because it has no immediate effect on the customer-perceived quality. To counter this problem, continuous improvement and control is required not only on the process but also on the product level. This enables us to proactively work against the otherwise inevitable quality decay.

Therefore, we proposed to use the same analogy of the PDCA cycle to describe the quality control process for software quality: *continuous software quality control* [48]. Figure 4.2 systematically shows this control process as a feedback loop. Starting from general product goals, we use a quality model, our central knowledge base about product quality, to specify quality requirements. Then, we give them

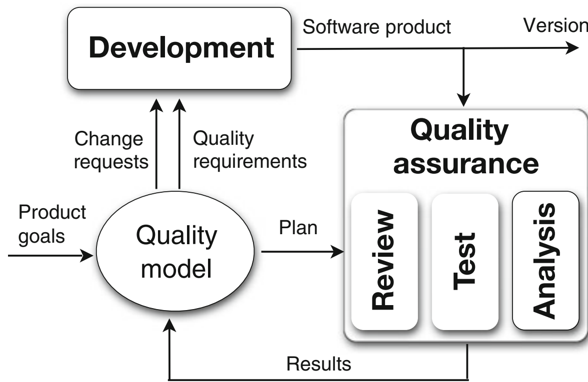


Fig. 4.2 Quality control loop

to the development. They build a version of the software product that is given to the quality assurance team. They apply various quality assurance techniques based on what the quality model proposes to measure the specified quality requirements. This includes reviews, tests and other analyses. The results are fed back into the quality model and compared with the quality requirements. Deviations together with changed product goals lead to change requests for the development team who produce a new version of the product. The cycle continues.

Ideally, we perform this cycle as often as possible, for example, together with the nightly build and continuous integration. As not all of the quality assurance techniques are automatic, however, the cycle time of the complete feedback loop cannot be 1 day. We need to find a compromise between the effort to be spent for performing quality assurance and the gain from avoiding quality decay. Therefore, there need to be different cycle times depending on the quality assurance techniques applied in that cycle. Many static analyses can be performed autonomously and comparably fast so that they can be executed at least with the nightly build or even at every check-in. The execution of automated tests should be part of the continuous integration but long-running tests should also only be executed with lower frequency. Manual analyses, such as code reviews, are only feasible at certain milestones. The milestones should contain defined quality gates that describe the expected level of quality so that the project achieves the milestone. For such important points in the project, the corresponding effort should be spent.

Practice shows that two different cycle frequencies with different analysis depths work best to prevent quality decay without decreasing development productivity (Fig. 4.3):

1. We execute automatic quality assurance on a daily basis, e.g. during the *nightly build*. We summarise the results of these analyses on a quality dashboard (see below). This dashboard provides all project participants (including the quality engineer and project managers) with timely information and gives developers

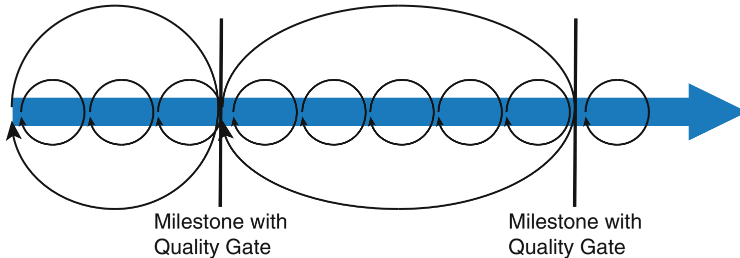


Fig. 4.3 Two cycle frequencies

the opportunity to identify and correct quality defects immediately. Trend data provided by the dashboard supports the quality engineer in identifying negative trends over time.

2. In a second cycle, we execute higher effort quality assurance techniques roughly every month, at least at each important milestone with a quality gate. These quality evaluation uses the data provided by the dashboards but includes additional analyses that require manual work. We investigate changes in trends to ensure that reasons for improvement or decline are fully understood. These detailed quality evaluations should result in quality reports that summarise the findings of the evaluation and illustrates general trends. These reports serve a reliable documentation of the status quo that can be communicated to management and used as reference for future discussions about trends in quality.

4.1.2 Tool Support and Dashboards

We have already discussed that automation is key for making continuous quality control a success. Only if we have a high level of automation, the higher quality control frequencies are possible. Hence, we need tool support. This includes:

1. Tool support for executing the actual quality assurance techniques, such as static analysis tools or automated tests.
2. A tool to combine the results and present to us the results, ideally already interpreted as a quality evaluation.

The probably most intensively used tool support for quality assurance today is automated tests. Not only unit test frameworks have made automating them highly popular but also other automation possibilities, such as capture/replay tools to automate GUI testing, are becoming popular. All these test automation tools are helpful for supporting quality control. As a result, we get the numbers of successful and failed test cases as well as test coverage. We will discuss testing in detail in Sect. 4.5. Static analysis is the second area that lends itself to automation. There is some tool support for manual reviews which can help to collect data about it. Even

with this tool support, reviews will probably not be conducted with high frequency. Automated static analysis, however, works perfectly with high frequency in quality control. Bug pattern tools or architecture conformance analysis tools can deliver warnings about various problems; metrics tools and clone detection tools can collect measurements that give us a better understanding of the state of the quality of our software. We will discuss static analysis in detail in Sect. 4.4.

For combining all these singular results, we usually apply tools called *dashboards*. Similar as in other areas, these dashboards have the aim to provide us with an overview of our product or project. A dashboard does not have to be about quality alone, but we will concentrate on quality dashboards here. The three main tasks of a dashboard in this context are *integration*, *aggregation* and *visualisation*.

With integration, we mean that the dashboard is a central tool that is able to start the execution of other tools so we do not have to do that manually. Ideally, the dashboard is a kind of command-line tool so that it is easy to integrate in our regular build and continuous integration systems. We have used the dashboard tool ConQAT[1] in most of our projects, because people involved in these projects develop this tool. It is freely available and very powerful. It allows a high level of customisation and, thereby, it integrates test tools such as JUnit or static analysis tools such as FindBugs. Some analyses, such as clone detection, are directly implemented in ConQAT.

The second task, the dashboard should support, is the aggregation of the results of the various sources of data, i.e. tests, static analysis and manual reviews. Depending on the data we receive, it is either already system-wide or specific for modules or classes. Module-specific measures need to be aggregated to higher levels, such as packages and system. The dashboard should support to aggregate to different levels and allow the users of the dashboard to choose the level they are most interested in. For example, when we measure the fan-in and fan-out of a class using ConQAT, we then want to see average values for the package level and system level. We introduced the possibilities of aggregation in Sect. 2.2.2. In addition, it is helpful if the dashboard can already automate the interpretation of the measures in some kind of quality evaluation. In the Quamoco approach to quality evaluation (Sect. 4.2), we generate a suitable configuration for ConQAT to do that.

Finally, the dashboard should not only create numbers from the data sources and aggregation but support the dashboard users by visualising the outcomes. Even if we employ a very structured, goal-driven measurement program, the results can be huge. For large systems, it is easy to get lost in the amount of data we get. Humans are visual beings, and visualisation helps to get an overview and to find problems.

A visualisation that often can be a crude representation of the data but which is useful for a very quick overview is traffic lights. Formally speaking, it is usually a rescaling aggregation. Some complicated data in interval or ratio is aggregated into an ordinal scale of red, yellow and green. Green usually represents the values that are "good", i.e. on an accepted level, yellow are problematic values and red are

[1] http://www.conqat.org

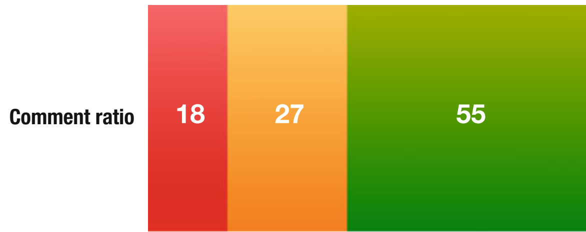


Fig. 4.4 Visualisation as traffic lights

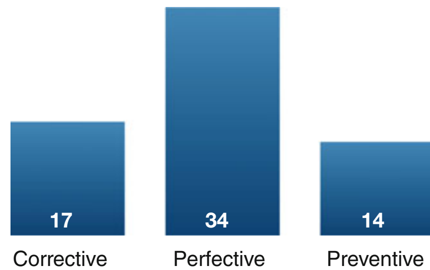


Fig. 4.5 Visualisation as bar chart

"bad" values. The example in Fig. 4.4 shows the percentage of classes in a software system with comment ratios on the red, yellow and green level. Red can be defined as a comment ratio between 0 and 0.1, yellow as 0.1–0.2 and red above 0.2. Another use of traffic lights with only red and green is to show the percentage of unit tests that passed and failed.

A bar chart is a useful visualisation for comparing different values. The example in Fig. 4.5 compares the number of change requests for each type of change request. This shows us whether we mostly fix bugs (corrective) or if we are able to build new features (perfective) and restructure and improve the internals of the system (preventive).

Trend charts are visualisations that help to track measures over time. In many cases, I am interested not only in the current state of my system but also in the change over time. This is especially useful for measures that indicate problems, such as clone coverage or warnings from bug pattern tools. Often it is not possible and also not advisable to make immediate huge changes to the system to remove all these problems. Nevertheless, it makes sense to observe these measures over time and to make sure that at least they do not increase or even better that they decrease. The example shown in Fig. 4.6 is the percentage of clone coverage of a system over 6 month.

Finally, tree maps are mostly used for finding hot spots in the system. The intensity of the colour indicates the intensity of the measure. The example in Fig. 4.7 is a tree map for clone coverage created by ConQAT. As this is a probability, it is in the range 0–1. The visualisation is then white for 0 and completely red for 1. Any shades in between show different levels of clone coverage. The tree map has the additional dimension of the squares it shows. These squares can show another measure, usually the size of classes or components.

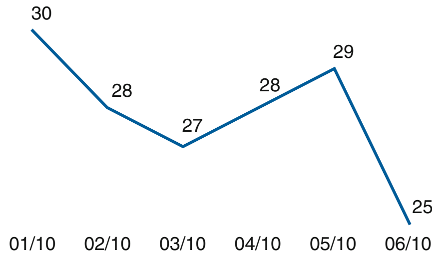


Fig. 4.6 Visualisation as trend chart

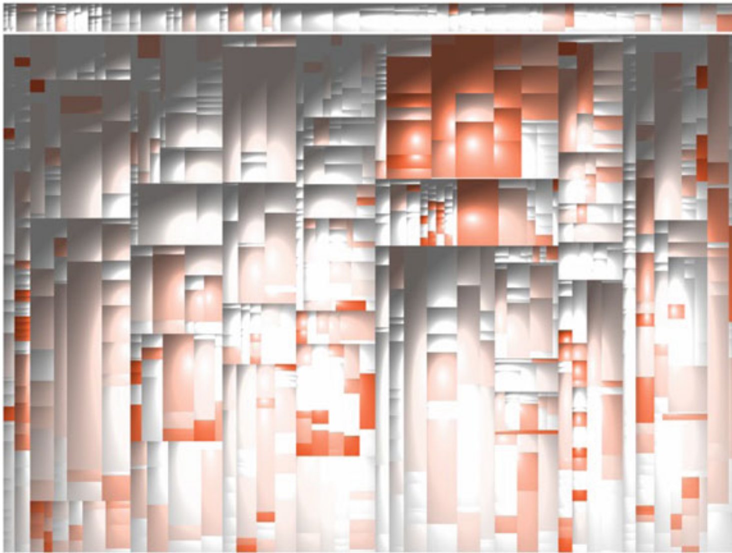


Fig. 4.7 Visualisation as tree map

4.1.3 Summary

In summary, we apply continuous quality control for the product quality of our software systems to counteract on the otherwise inevitable quality decay. We perform various quality assurance techniques in their respectively appropriate cycle frequencies to get a continuous feedback on the quality level of our product and are therefore able to react in a timely manner. Automation is key to make this quality control continuous, and especially a dashboard tool can help us to integrate, aggregate and visualise the results. How we perform the necessary analyses and quality evaluations is the subject of the remainder of this chapter.

4.2 Quality Evaluation and Measurement

The objective of the Quamoco quality modelling approach (see Sect. 2.4) was to make quality more concrete. We wanted to break it down to measurable properties to get a clear understanding of the diffuse quality characteristics we usually have to deal with. The resulting quality model structure allows us to do this now: collect data to measure product factors and aggregate the results up to quality aspects. Why do we need such a quality evaluation? It gives us a variety of possibilities: We can verify if our product conforms to its quality goals. We can find problematic quality aspects and drill down in the results to find improvement opportunities. We can compare different systems or sub-systems, for example, when we have to select a product to buy. We could even use this as the basis for certifying a certain level of quality. Figure 4.8 shows an overview of how this quality evaluation is performed.

4.2.1 Four Steps

Conceptually, there are only four steps for getting from the software product to a complete quality evaluation result:

1. Collect data for all the measures from the software product.
2. Evaluate the degree of presence of the product factors based on the measure values on a scale from 0 to 1 for each product factor.
3. Aggregate the product factor evaluations along the impacts to quality aspects using weights.
4. Aggregate further along the quality aspect hierarchy and interpret the results using a mapping to grades.

What do we need in the quality models to make this work? We have already discussed measures and their concrete measurement descriptions for specific programming languages or technologies in instruments. Therefore, we can use them directly to collect values for the measures. In addition, there is a model element called *evaluation* for each factor, i.e. all product factors and quality aspects. In the evaluation we can describe how we transform the results from the lower levels into an evaluation of the factor. Hence, for product factors, the evaluation describes how the values for its associated measures are combined to come to its evaluation. For quality aspects, the evaluations of lower-level product factors and/or quality aspects are aggregated to an evaluation of that quality aspect. In principle, these evaluation specifications could only be textual, giving a description for a human to make the evaluation. As there are hundreds of such evaluations in realistic quality models, we need to automate that. Therefore, we have machine-readable evaluation specifications.

For product factors, we call them *measure evaluations*, because they specify the calculation for getting the product factor evaluation from the values of the measures. Let us look at the example of the product factor [Source Code Part | DUPLICATION].

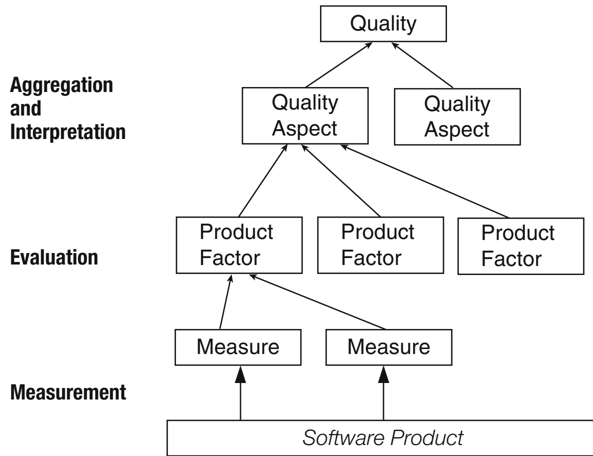


Fig. 4.8 Overview of the evaluation approach [211]

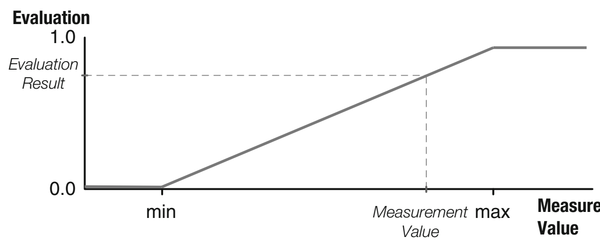


Fig. 4.9 Example of a linear evaluation function [211]

It describes that "source code is duplicated if it is syntactically or semantically identical with the source code already existing elsewhere". We concentrate on measuring the syntactically identical code with the measure *clone coverage*. It describes the probability that a randomly chosen code statement is duplicated in the system. Hence, the higher the clone coverage, the more the product factor [Source Code Part | DUPLICATION] is present in the system. One could now start to assign different levels of clone coverage to the evaluation or just directly calculate it by the equation "1−clone coverage". To make such calculation more uniform and to express that there are usually two thresholds – a lower acceptable boundary and an upper acceptable boundary – we use special linear functions. Figure 4.9 shows an example.

The intuition is that there are two thresholds. Above and below these, I do not care about the detailed measure value, but the product factor is completely present or not present at all. Between these threshold values (min and max), the evaluation increases linearly. For the [Source Code Part | DUPLICATION] example and the measure *code coverage*, this means that below a certain threshold, say 0.05, I judge that there is no duplication in the source code. Above the max threshold, say 0.5, I consider the

source code completely duplicated. Between min and max, the evaluation increases. Therefore, I can find for a certain measurement value for a concrete system an evaluation result between 0 and 1. Depending on the measures and product factors, the linear function could be the other way round.

This is rather easy for *code coverage*, because it is already a measure on the system level, i.e. it describes a property of the whole system. We have many other measures, however, that are not system-level measures. We use many rules from static analysis tools which single findings. We can sum up these findings, but it is still not the same if I have 10 findings in a system with 1,000 LOC or 10,000,000 LOC. What we need is a *normalisation* of the measurements with respect to the system. For that, we either set the LOC of the affected system part in relation to the LOC of the complete system or the number of affected entities in relation to the number of entities. A finding about missing constructors in classes can be summed and divided by the total number of classes in the system. A finding about a more specific library method, such as a missing closing of a database connection, is easier normalised by LOC. We bring all measure to the system level so that we can combine them more easily in the measure evaluations.

The evaluation specification is simpler for quality aspects. A quality aspect has either a set of product factors that impact them, other quality aspects that refine them or a mixture thereof. We chose again a simple unification of these cases by using a weighted sum. Hence, each impact gets a weight that describes how much of the complete evaluation of a quality aspect is driven by this product factor or quality aspect. We then sum all weights to calculate the weighting factor for each product factor or quality aspect. We multiply each evaluation result with its corresponding weighting factor and add up all of these to the evaluation for the quality aspect. The only thing we need to consider is whether impacts from product factors are positive or negative. A negative impact means that we use 1−the evaluation result from the product factor. For example, there are many impacts onto the quality aspect *analysability*. Let us assume there are only three:

- [Source Code Part | DUPLICATION] $\xrightarrow{-}$ [Analysability], weight 10, evaluation 0.2
- [Source Code Part | USELESSNESS] $\xrightarrow{-}$ [Analysability], weight 5, evaluation 0.3
- [Source Code Part | DOCUMENTEDNESS] $\xrightarrow{+}$ [Analysability], weight 7, evaluation 0.5

Hence, the weighting factor is 22, the sum of the weights. For the evaluation result for analysability, we calculate $10/22 \cdot 0.2 + 5/22 \cdot 0.3 + 7/22 \cdot 0.5 = 0.32$. A simple sum adjusted by the weights gives us again the degree of presence of the quality aspect. Analysability is present to a third. This is a good first indication but hard to interpret: Is that good or bad?

4.2.2 *Interpretation Schema*

Therefore, we also introduced an interpretation schema for quality aspects. The intuition is that we judge software quality similar to a dictation in school. You have to write a text read out to you, and afterwards the teacher counts the mistakes. The

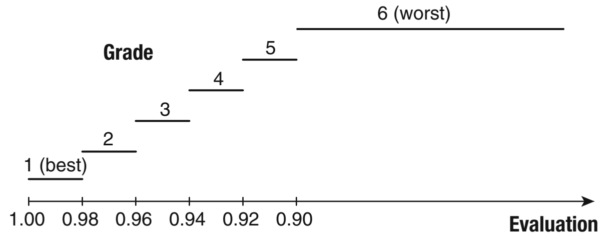


Fig. 4.10 Interpretation as grades [211]

more mistakes, the worse the grade you get. If you want a really good grade, your text needs to be almost flawless. This is how we interpret the evaluation results between 0 and 1. An example interpretation using German school grades is shown in Fig. 4.10. In our previous example, the analysability with 0.32 would clearly result in a 6, the worst grade. You should definitely do something to improve it.

The final question, you probably have, is where we get the thresholds for the evaluation specifications and interpretations from. We call this *model calibration* because we calibrate these values usually according to empirical data. You can also calibrate each specification by hand using your own expert opinion. This is, however, difficult to repeat and to explain. Therefore, we rely on a more objective method: We apply the measures to a large amount of existing systems to understand what is "normal". The normal range defines then the upper and lower thresholds. You can find more details about the calibration in [211].

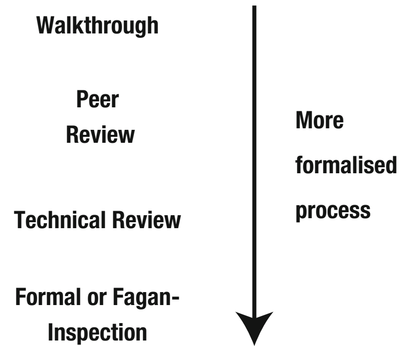
4.3 Walk-Throughs, Reviews and Inspections

Structured reading of artefact content is one of the most important quality assurance techniques as it enables one to detect problems that might lead to failures as well as problems that make it harder to comprehend the content. No other technique is as effective as reviews in finding this variety of problems. Depending on the exact execution of the reading, this technique is called walk-through, review or inspection. Unfortunately, it is not used frequently in practice. Ciolkowski et al. [39] found in a survey that less than half of the respondents perform requirements reviews and only 28 % do code reviews. Major obstacles stated were time pressure, cost and lack of training. To overcome these obstacles, the following gives you an overview of the available techniques, insights into the actual effectiveness and efficiency of reviews as well as suggestions on automation for reducing costs and integrating reviews into continuous quality control.

4.3.1 Different Techniques

Because there are many variations of walk-throughs, reviews and inspections the terminology differs in existing literature and practical use. We give one possible

Fig. 4.11 The main review techniques



terminology that fits to most usages of terms we have encountered, but you may find other uses easily. The most important differentiation between different review techniques is the formalisation of the review process. As Fig. 4.11 shows, walk-throughs are the least formalised, while formal inspections have the most formalised process. This means in practice that walk-throughs can be conducted in various ways. An inspection, in contrast, has clear process steps that have to be followed and that should be fixed in the company-wide development process.

The boundaries between these different techniques is blurry and one technique might borrow specific parts from another to make it more suitable for the task at hand. Most variation in these techniques can be explained along four dimensions that Laitenberger [132] identified: process, roles, artefacts and reading techniques. The process defines the steps of the review and which steps are mandatory. The participating roles determine who has to take which part in the review. The artefacts that are reviewed, including support artefacts such as checklists or requirements, prescribe when in the process it is possible to use the review, because the artefacts have to be created first. Finally, the used reading techniques differentiate how the reviewers conduct their reading which also has influences on the required artefacts.

Walk-throughs, also called presentation reviews, have the aim to give all participants an understanding of the contents of the analysed artefact. The author guides a group of people through a document and his or her thought processes to create a common understanding. It ends with a consensus on how to change the document. For that we need only two roles in the walk-through process: the author and listeners. The solely defined process step is the explanation of the document by the author. How the consensus is reached is usually left open. A walk-through is possible for all kinds of documents. Specific reading techniques are not used.

In *peer reviews*, the authors do not explain the artefact. They give it to one or more colleagues who read it and give feedback. The aim is to find defects and get comments on the style. Similar to walk-throughs, peer reviews make use of only two roles, the *author* and *peers*, usually colleagues that are organisationally close to the author. There are two process steps: (1) individual checking by the peers and (2) feedback from the peers to the author. The peers often give the feedback informally in a personal discussion, but they can also write it down. It is possible

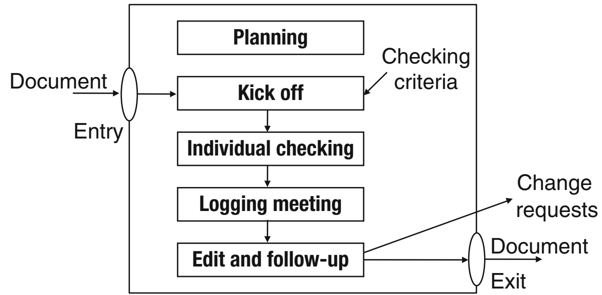


Fig. 4.12 The inspection process (derived from [73])

to use different reading techniques in the individual checking step, but it is usually not demanded by the process. The author could ask the peers, however, to look for specific aspects. You can use a peer review for any kind of document.

Technical reviews further formalise the review process. They are often also management reviews or project status reviews with the aim to make decisions about the project progress. In general, a group discusses the artefacts and makes a decision about the content. The variation in the process and roles is high in practice. There are at least the two process steps individual checking and joint discussion. The involved roles depend on the type of artefact that is reviewed. For example, in a design specification, implementors as well as testers should be involved. Again, different reading techniques can be used, but most often, it is checklist-based reading.

Inspections have the most formalised process which was initially proposed by the software inspection pioneer Fagan [59]. They contain formal individual and group checking using different sources and standards with detailed and specific rules how to check and how to report findings. Furthermore, we distinguish between the roles *author*, *inspection leader*, *inspector* and *scribe*. The author, however, does not take part in the actual inspection but is only involved to answer questions and correct minor problems directly after the inspection. The leader is the moderator and organiser of the inspection. The inspectors read and report findings, while the scribe documents them.

The process of an inspection consists of the steps *planning*, *kick off*, *individual checking*, *logging meeting* and *edit and follow-up* (Fig. 4.12). In the planning step, the inspection leader, or moderator, organises a particular inspection when artefacts pass the entry criteria to the inspection. The leader needs to select the participants, assign them to roles and schedule the meetings. The kick off meeting, sometimes called overview step, is not compulsory but is especially useful if inspections are not well established or there are many new members in the inspection team. This phase is intended to explain the artefact and its relationships to the inspectors. It can be helpful for very complex artefacts or for early documents such as requirements descriptions.

The next step, individual checking, is the main part of the inspection. The inspectors read the document based on predefined checking criteria. Depending on these criteria, they need to use different reading techniques. The most common

technique is checklist-based reading. The checklist contains common mistakes and important aspects that the inspectors need to look for. You can find, however, a variety of alternative reading techniques in the literature. Usage-based reading [196] proposes to employ use cases as a guide through the document, defect-based reading to define and then look for specific types of defects and perspective-based reading [9] emphasises reading from the point of view of different stakeholders of the document, e.g. developers, testers or customers.

The logging meeting is the next step in the inspection process. All inspectors come together to report their findings which are logged by the scribe. The author may take part to answer open questions of the inspectors. In some instances, here the document is read again together, but most commonly, only the found issues are collected. The main benefit of the meeting is that the collective discussion of findings reveals new issues as well as establishes a consensus on the found problems. The logging meeting closes with a joint decision whether the artefact under inspection is accepted or needs to be reworked. In the edit and follow-up step, the author removes minor problems directly from the document. For larger issues, the inspection leader creates change requests. This step ensures that the document conforms to the exit criteria. Inspections are possible for all kinds of documents.

4.3.2 Effectiveness and Efficiency

In practice, reviews are often considered less important than testing [212]. This is in contrast, however, to the findings we have that reviews are effective and efficient in quality assurance. Although research has left some questions open, there is a considerable body of knowledge that you can make use of in your quality control loop. Unfortunately, the studies do not always clearly distinguish between walk-throughs, peer reviews, technical reviews and inspections. Most studies, however, investigate more formalised review techniques such as technical reviews and inspections.

We found by analysing a variety of studies and reports from practice that on average about a third of the defects in an artefact is found by reviews [200]. The variance of the effectiveness, however, is large. It can go up to 90 % if the review is conducted correctly. This means that there is a defined process and that the reading speed is about one page per hour. If we read less, we do not find more defects, but if we read more than two pages per hour, we miss defects.

Despite the large conceived effort needed for reviews – several people need to read the same documents – they are efficient. On average, you find 1–2 defects per person-hour including all preparation and meeting efforts. This is at the same level as testing. Similarly to the effectiveness, efficiency can be lifted even higher. High-quality inspections find up to six defects per person-hour which is hardly achievable with testing. Moreover, most studies show that tests and reviews find (partly) different defects [184]. Therefore, to find as many defects as possible before you ship the software to your customer, you need to perform reviews in addition to

tests. This is also stated concisely in the Hetzel–Myers law [57]: "A combination of different V&V methods outperforms any single method alone".

Moreover, reviews are an extraordinary technique as they are generic and therefore very flexible in terms of when and on what you can use them. In particular, they have the huge advantage over most other quality assurance techniques that you can employ them early in the development on prose documents such as requirements specifications. Hence, you can find and remove the defects when it is still comparably cheap. Furthermore, in comparison to dynamic techniques such as testing, a review gives the exact defect localisation. A reviewer identifies a problem and can point the author to the exact place where it occurs. A tester can only show wrong behaviour. We found in the existing studies that the effort for removing defects found in a requirements review is on average one person-hour per defect, 2.3 person-hours per defect in a design review and 2.7 person-hours per defect in a code review. Hence, as rule of thumb, you have to spend 1 person-hour for a requirements defect, 2 person-hours for a design defect and 3 person-hours for a code defect. These results strongly encourage to perform a lot of reviews early in the development to save later on. As anything, this has a break-even point where it is not beneficial anymore to further invest into early reviews. To instantiate early reviews at all, however, is always beneficial.

Finally, the soft benefits of reviews are not to be underestimated. A review always means that different people from the development project read documents and then sit together to discuss them. Ideally, these people represent different roles and different levels of expertise. This way, the review also helps to create a common understanding between different stakeholders, the developers and the testers, for example. Furthermore, it serves to transfer knowledge from the more experienced team members to people new in the project or the profession. Nobody has been able to quantify this effect, but our experiences and those of our collaborates suggest that it is huge. If none of the data has convinced you to introduce reviews, this is a strong argument to do so.

4.3.3 Automation

Automation is crucial for including such a strongly manual task as reviews in continuous quality control. Depending on the concrete review technique you use, different tool support is possible. Furthermore, different aspects of the review process can be automated. Tenhunen and Sajaniemi [195] identified four major areas that can be automated:

- Material handling (all document-related issues such as file format, versions or navigation)
- Comment handling (all comment-related functions, such as adding, viewing, reviewing, reporting or summarising)
- Support for process (features that automate or otherwise benefit the inspection process)
- Interfaces (use of external software or other tools)

Any review technique can benefit from automation in material and comment handling. Having both only on paper hampers quick analyses and distributed checking. In modern distributed development, it is often necessary to have reviewers around the globe that need quick access to the material. The tool-supported handling of comments allows the author to have quick access to the comments to incorporate them. If the tool supports further aspects of the process, it is necessary that these are clearly defined. Then, however, many benefits can arise such as automatic distribution of reviews to suitable reviewers or collection of review performance data that the project manager can analyse to improve the review process. These data then can also be fed into the continuous quality control process. This would be an interface to the corresponding tool support for quality control. But interfaces to further tools such as static analysis tools (Sect. 4.4) can also be beneficial. The warnings produced by the analysis tool can be shown to the reviewer in-line with the code to be reviewed. This helps the reviewer not to miss any problematic aspects.

For example, the Android Open Source Project[2] uses a web-based review tool for their peer reviews. They use Gerrit,[3] also an open source tool, that is based on the Git configuration and version management system. The philosophy in the Android project is that any change needs to be approved by a senior developer who checks out the change and tests whether Android still compiles and does not crash, i.e. the developer performs tests. The confirmation is handled within Gerrit. Moreover and more interesting for reviews, each change also has to be read by at least two reviewers. The list of approvers and reviewers for each change can be seen within the Gerrit web environment. The reviewers see the complete change set, i.e. all changed files, and can read the changes either in a side-by-side compare view or in one unified view. They have the possibility to add comments to any line in the changed files and start discussions about these comments. Finally, they can approve or disapprove of the changes. All team members get an immediate overview of all changes, comments and review tasks. Another example is the open source review manager *RevAger*,[4] which supports in contrast to Gerrit the traditional face-to-face logging meeting but automates many aspects around this meeting and is especially efficient in logging the found issues.

4.3.4 Usage

Reviews are essential in the quality control loop to analyse factors in the artefacts that cannot be automatically assessed. This includes many aspects that have a semantic nature, i.e. that depend on the understanding what the content of the artefact means. For example, the appropriateness of code comments cannot be

[2] http://review.source.android.com

[3] http://code.google.com/p/gerrit/

[4] http://www.revager.org

assessed completely automatically. There are some automatic checks that give indications, but if you want to know whether the comment fits to what is done in the code, someone has to read both and compare them. Hence, there are many factors that have to be evaluated in the quality control loop that need reviews. For each work product in your development process, you should therefore plan at least a walk-through. Critical documents should be inspected.

Hence, reviews are indispensable to assess what you have specified in your quality model (see Sect. 2.4). Especially, if you invested in building detailed and concrete quality models following our Quamoco approach, you can and should exploit this by generating the checklists for your reviews directly from the quality model. The tool support we discussed in Sect. 2.4 is able to do that. This gives you two benefits: (1) the guidelines are streamlined with the quality goals and requirements in your quality model, and (2) you reduce redundancies between the quality model and the checklists.

Nevertheless, reviews need considerable manual effort. Therefore, they cannot be performed in a daily build. The frequency of reviews should be determined by other factors such as the finalisation of, or changes in, a specific document. For example, you should perform a review after you finished a requirements specification or the source code of a Java class. You should persist the information about quality problems that the review generated to use it in the quality control loop. Even if the information is not always collected freshly, it is available at any time. Furthermore, the information which parts of the system have been reviewed itself is interesting for quality control and should be collected and visualised.

4.3.5 Checklist

- Have you made sure that reviews are done in an atmosphere of joint improvement instead of blaming the author?
- Is there a planned review (at least a walk-through) for any relevant document in your development process?
- Have you identified critical documents and scheduled a formal inspection?
- Have you made clear, for each review to be conducted, what type of review it should be, who has to participate and how it is going to be conducted?
- Do you make sure that the reviewers have access to all other documents that they might need during their checking (requirements specifications, checklists, design documents)?
- Have you made sure that the used other documents are derived from and are in sync with your overall functional and quality goals?
- Do you derive checklists automatically from your quality models to reduce redundancy?
- Do you adhere to the optimal reading speed of about one page per hour?
- Do you have a defined infrastructure that saves the results of the review as well as other data that was collected?

- Do you use tool support wherever possible in the review process?
- Have you incorporated the review results in overall (continuous) quality analyses?
- Have you modelled your existing checklists in a quality model and do you use this model for generating (role-specific) checklists?

4.3.6 Further Readings

- Gilb and Graham [73]
  This is the classic book about inspections. If you are new to inspections and plan to introduce them to your development process, I highly suggest you to read it. It gives a lot of practical guidance and forms you can use. Because of its age, the book does not include up-to-date information about automation.
- Shull et al. [189]
  This is a gentle introduction to perspective-based reading, a specific reading approach, which you can use in your inspections. This article explains the approach and discusses experiences.

4.4 Static Analysis Tools

Static analysis tools are programs that aim to find defects in code by static analysis similarly to a compiler. The results of using such a tool are, however, not always real defects but can be seen as a warning that a piece of code is questionable in some way. Hence, the differentiation of true and false positives is essential when using static analysis tools. There are various approaches to identify critical code pieces statically. The most common one is to define typical bug patterns that are derived from experience and published common pitfalls for the programming language. Furthermore, the tools can check coding guidelines and standards to improve readability. Finally, more sophisticated analysis techniques based on the data flow and control flow find, for example, unused code or buffer overflows. We will give an overview of these techniques and discuss clone detection in more detail, because it proved very useful in our practical experiences.

4.4.1 Different Tools

Any tool that does not execute the software for analysing it but uses the source code and other artefacts belongs to the class of *static analysis tools*. Hence, the differences between the tools at the ends of the spectrum are huge. The rough

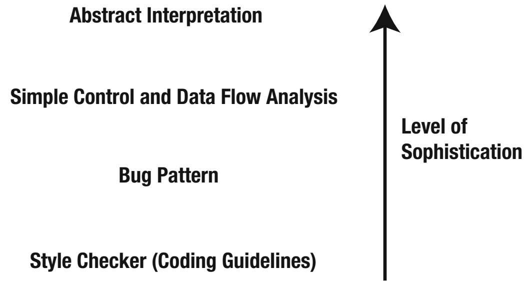


Fig. 4.13 Classification of static analysis tools

classification of Fig. 4.13 calls one end of this spectrum *style checkers*. These tools analyse the source code of a system for conformance with predefined coding guidelines. The analysis includes formatting rules, the casing of identifiers or organisation-specific naming conventions. These analyses are comparably simple and the tools can automatically resolve many problems they find. This functionality is basic enough so that integrated development environments (IDE) often contain this kind of checks. Eclipse, for example, can directly detect if the code does not conform to a defined format in the standard Java development edition. Another style checker for Java is Checkstyle[5] that in addition can check, for example, that you do not use more than a specified number of statements in a method or Boolean operators in an expression.

The most common type of static analysis tools is *bug pattern tools*. They formalise common pitfalls and anti-patterns for a specific programming language. For example, in Java, strings should be compared using the *equals()* method instead of ==. The latter would only check whether the compared variables point to the same object, not if they contain the same strings. Although in some cases the comparison of object IDs is the intended behaviour, in most cases, it constitutes a mistake by the programmer. This likely defect is found by a bug pattern tool which issues a warning to the programmer.

Most types of defects found by bug pattern tools are similar to the string comparison example. The bug patterns need to be described in purely syntactical terms; the semantics cannot be analysed. A run of such tools is quick and computationally cheap. Bug pattern tools are available for virtually any programming language. The availability, however, differs strongly: for Java, there are several usable open source tools such as FindBugs[6] and PMD[7], and for C, the prevalent tools in practice are mostly commercial such as PC-lint[8] and Klocwork Insight.[9]

---

[5] http://checkstyle.sourceforge.net/

[6] http://findbugs.sourceforge.net/

[7] http://pmd.sourceforge.net/

[8] http://www.gimpel.com/

[9] http://www.klocwork.com/products/insight/index.php

A more sophisticated analysis than merely searching for syntactical patterns is control and data-flow analyses. Bug pattern tools often contain a small number of detectors that use the control or data flow to find possible defects. For example, FindBugs can detect *dead code*, private methods that are never used, which is a control-flow analysis. PC-lint employs data-flow analysis to find variables that are written but never read.

Abstract interpretation generalises these simple control and data-flow analyses to any abstraction of the code that is abstract enough for an automatic analysis but concrete enough to make precise statements about properties of the program. Tools such as Coverity[10] or PolySpace[11] find buffer overruns or pointer accesses outside valid memory.

4.4.2 Clone Detection

One of the static analysis techniques I want to emphasise in particular is clone detection. It is a powerful technique to get an impression of the ageing process of a software, and it is highly automated at the same time.

What Is a Clone?

A clone is a copy of a part of a software development artefact that appears more than once. Most of the clone detection today concentrates on code clones but cloning can happen in any artefact. In code, it is usually the result of a normal practice during programming: I realise that I have implemented something similar somewhere else. I copy that part of the code and adapt it so it fits my new requirements. So far it is not problematic, because we would expect now that the developer performs a refactoring to remove the introduced redundancy. Often, this does not happen, however, either because of time pressure or because the developer is not even aware that this can be a problem.

The developer does not keep an exact copy of the code piece but changes some identifiers or even adds or removes some lines of code. The notion of clones incorporates that. To identify something as a clone, we allow normalisation to some degree, such as different identifiers and reformatting. If complete statements have been changed, added or deleted, we speak of *gapped clones*. It is one of the parameters we have to calibrate in clone detection how large this gap should be allowed to be.

[10] http://www.coverity.com/

[11] http://www.mathworks.com/products/polyspace/

Impact of Cloning

It is still questioned today in research if cloning is really a problem [118]. From our studies and experiences from practice, cloning should clearly be avoided because of two reasons.

First, it is undeniable that the size of the software becomes larger than it needs to be. Every copy of code adds to this increase in size which could be avoided often by simple refactorings. There are border cases where a refactoring would add so much additional complexity that the positive effect of avoiding the clone would be compensated. In the vast majority of cases, however, a refactoring would support the readability of the code. The size of a software is correlated to the effort I need to spend to read, change, review and test it. Any method cloned needs an additional unit test case which I either also clone or have to write anew. The review effort increases massively, and the reviewers become frustrated because they have to read much similar code.

Second, we found that cloning can also lead to unnecessary faults. We conducted an empirical study [113] with several industrial as well as an open source system in which we particularly investigated the gapped clones in those systems. We reviewed all found gapped clones and checked whether the differences were intentional and whether they constitute a fault. We found that almost every other unintentional inconsistency (gap) between clones was a fault. This way, we identified 107 faults in five systems that have been in operation for several years. Hence, cloning is also a serious threat to program correctness.

Clone Detection Techniques

There are various techniques to detect clones in different artefacts [128]. Most of them are best suited to analyse source code but there are also approaches to investigate models [47] or requirements specifications [112].

We work mainly with the implementation of clone detection as provided in the tool ConQAT. It is proven as working under practical conditions, and it is now used at several companies to regularly check for cloning in their code. The measure we use to analyse cloning is predominantly *clone coverage* which describes the probability that a randomly chosen line of code exists more than once (as a clone) in the system. In our studies, we often found clone coverage values between 20 % and 30 % but also 70–80 % is not rare. The best code usually has single digit values in clone coverage.

In general, false positives are a big problem in static analysis. For clone detection, however, we have been able to get rid of this problem almost completely. It requires a small degree of calibration of the clone detection approach for a context but then the remaining false-positive rates are neglectable. ConQAT, for example, provides a blacklisting and can read regular expression to ignore code that should not be considered, such as copyright headers or generated code.

Finally, we use several of the visualisations the dashboard tool ConQAT provides to control cloning: a trend chart shows if cloning is increasing or a tree map shows us which parts of our systems are affect more or less strongly by cloning.

4.4.3 Effectiveness and Efficiency

In terms of ODC defect types (see Sidebar 3 on page 108), bug-finding tools are effective in finding two types of defects [220]: assignment and checking. Assignment defects are incorrect initialisations or assignments to variables. Checking defects are omissions of appropriate validation of data values before they are used. Despite all tools being effective in these defect types in general, different tools tend to find different specific defects because of their detection implementations.

Although the bug-finding tools are mostly reduced to these two defect types, they are able to find up to 80 % of the defects in a software product [200, 209]. The defects found are, however, mostly of low severity. For severe defects, the effectiveness is about 20 % [200, 209]. The most severe defects are of the types function and algorithm, i.e. constitute problems in the functionality and behaviour. These problems are very hard to detect using static analysis. In an industrial study, we could not find a single field failure that could have been prevented by using the static analysis tools [205]. In the same study, however, we found that code modules with a high number of warnings from different tools have a high probability to contain severe faults. Hence, even if the tools might not directly lead you to a fault that causes a field failure, you have a high chance to identify code modules that are badly written and hence contain such faults.

The effort needed to detect a defect is roughly in the same order of magnitude as for inspections [220]. A lot of the effort, however, goes into the installation and configuration of the tool. It is specifically important to adapt the tools to your environment, for example, your programming guidelines. With this adaptation, it is possible to reduce the false-positive rates to a tolerable level. Non-adapted static analysis can result in false-positive rates of up to 96 % [209]. After adaptation, the defect detection is quick and cheap and results on average in less than half a person-hour per defect [205]. On top of the effort, there can be additional licence fees for the tools which are significant for most commercial tools. Depending on the programming language, however, there are often very useful open source alternatives.

4.4.4 Usage

Static analysis tools comprise an important part of the quality control loop, because they are so cheap to run and rerun. Hence, in the moment the first code is written, the tools can be employed in the quality control loop and provide up-to-date information

after each run in the continuous build of the software. The main challenge is the suitable configuration and adaptation of the tools to make them work efficiently in your environment. This includes the installation of the tools at the programmer's work stations and the build server, including it in the build system, connecting it to the quality dashboard, removing unsuitable checks and adding specific checks.

Installing the tools on the work stations of the programmers helps them to identify problems before they go into the source code repository. The ideal case is to have the tools integrated in the programming IDEs of the developers, so that they get immediate feedback if a tool finds a problem. In that case, you need to establish versioning of the configurations of the tools so that all developers work with the same configurations. In addition, running the tools also with the continuous build helps to double-check the rules and to aggregate and visualise the results.

If you maintain an existing code base for which you have not used static analysis tools so far, you will get an overwhelming number of warnings after their first run. It is not advisable to immediately change the code to remove all these warnings as this large change would be very costly and risky. Instead a proven approach is to run the checks and watch the trend over time. Even if you cannot remove all warnings now, you should take care not to increase the number of warnings. The existing warnings can then be removed over time in connection with other changes such as the introduction of new features.

Finally, although static analysis does not often find problems that would lead directly to a field failure, it still can help in preventing field failures. There is indication that the modules with a high number of warnings from static analysis tools are also the modules that contain faults that will lead to field failures [205,220]. Hence, you should aggregate these warnings also in the dashboard to perform a hot spot analysis of problematic modules that then should be inspected or tested more intensely.

4.4.5 Checklist

- Have you configured the static analysis tools so that they fit to your quality model?
- Have you discussed the requirements for the tools with your team?
- Have you adapted the used static analysis tools to further requirements of your organisation, product and project?
- Do you employ more than one tool to maximise defect detection?
- Do you use at least style checkers and bug pattern tools for all your code?
- Do you use data-flow and control-flow analysis tools for critical code?
- Have you integrated the execution and data collection of the static analysis tools into an overall quality dashboard?
- Do you watch the trend analysis of the warnings issued by the static analysis tools?

- Do you aggregate the warnings of the static analysis tools to identify problematic hot spots in your software product?
- Do you give the programmers access to the warnings directly in their IDEs if the tools allow this?

4.4.6 Further Readings

- Ayewah et al. [5]
  An easy to read and general introduction to bug pattern tools with a focus on FindBugs.
- Chess and West [33]
  This book is from the people behind the tool Fortify. They describe here specifically how to use that tool to support security.
- Bessey et al. [19]
  An interesting behind-the-scene discussion of the makers of Coverity on how to build static analysis tools and also how to build a company around it.

4.5 Testing

With *testing*, we mean all techniques that execute the software system and compare the observed behaviour with the expected behaviour. Hence, testing is always a dynamic quality assurance technique. Testers define test cases consisting of inputs and the expected outputs to systematically analyse all requirements. We give an overview of the variety of test techniques, discuss their effectiveness and efficiency as well as how they can be automated. Using this knowledge, we describe how they are included in the quality control loop.

4.5.1 Regression Testing

The basis of our approach to software quality control is to employ the control loop (Sect. 4.1) in the iterative development of the software system. Thereby, we identify defects early and soon after they were introduced. This means we also run all tests in each quality control loop to check whether the new code works and the old code is not broken. This is commonly called *regression testing*. It is a test approach in which we reapply test cases after changes. The idea is that once the test suite (i.e. a set of test cases) is developed, it can be run again after a change of the system was done (either bug fix or new feature) to make sure that existing functionality and quality has not been broken by the change. All test techniques we will discuss later on can provide test cases to be used in regression.

Already in initial development but even more so for successful maintenance, comprehensive regression testing is imperative. Making changes without regression tests is suicidal, because you can break the system without even realising it. It is not feasible, however, to do manual tests every time you commit a change to the system. Hence, regression testing must be automated as much as possible. Besides the effort and time savings, there are more reasons for this automation:

- Automated tests ensure repeatability.
- Manual tests are almost always underspecified and therefore executing the same manual tests multiple times creates very different traces in the system.

Having these automated tests, they must be run as often as possible, e.g. every night. This ensures that we catch defects early when they are still cheap and close to when they were introduced which eases debugging. Furthermore, you cannot tolerate failing tests in these nightly tests as new failures will hide in the failing tests. You will miss them, and they will then cause more trouble and cost more than necessary. Regression testing reveals failures. The causing faults, however, are often hard to find. One way to tackle this is to shorten the test cycles and run regression tests after each change, but this is not possible for large test suites. In addition, it is also the reason that regression testing should be done on all levels: unit, integration and system. On the unit or integration level, the scope of the test is smaller and this can help you in locating the fault to the failure. Having found the fault, you should use this information as a source for a new test case. You make sure that this kind of defect is definitely caught and will be easier to debug in the future.

Automated regression testing does not only require suitable test cases and a proper automation. Often it is equally hard to define a suitable test environment where, for example, the database can be reset to a defined status after each test (or collection of tests). In system testing, test execution often requires third-party systems that need to be present or mocked. While this can be set up for a singular system test session relatively easy, it is often non-trivial to create such an environment for tests that are run every night.

Unfortunately, despite its importance in modern development processes and quality control, regression testing is underrepresented in academic research. Also many textbooks discuss it only as a side aspect. I am convinced it should be the underpinning of all test efforts.

4.5.2 Different Techniques

Testing is the most important quality assurance technique in practice [212]. It is, however, not a single homogenous set of techniques, but it contains all structured executions of a software system with the aim to detect defects. The two major dimension we need to consider are the test phases and what drives the test case derivation [15, 74, 115, 161]. Figure 4.14 shows these two dimensions with concrete examples and how they can be placed according to those dimensions.

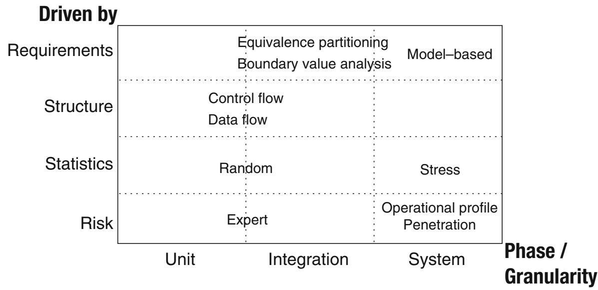


Fig. 4.14 The two major dimensions of test techniques [202]

On the phase dimension, we have the main parts *unit test* (also called *module* or *component test*), *integration test* and *system test*. In unit tests, the testers only analyse single components of the system in isolation using stubs to simulate the environment. During integration tests, they combine the components and analyse their interaction. Finally, in system testing, the testers check the whole system either in a test environment or already the production environment. Depending on the type of system, this can include a complex software/hardware/mechanics conglomerate. In addition, if the customer makes the system test, we call it *acceptance test*. To call this dimension *phase* can be misleading, as we do not expect to run each level of tests separately one after the other in each iteration. Therefore, you can also think "granularity" here: modules, groups of modules and complete system. We will cover each phase or granularity in detail in the following:

Phases

Unit Test

Any modern programming language supports decomposing a system into components or units. In object-oriented languages, these are usually classes; in other languages, there are modules. These units of a software system are the smallest building blocks that can reasonably be tested. A test on the statement level does normally not provide interesting results. A unit, however, describes a piece of functionality and thereby is the interaction of several statements. The correct implementation is tested by *unit testing* against the module specification.

Unit tests are developed at the same time as the units. In many companies, the developers of the units themselves specify and perform the unit tests. In test-driven development, the developers even write the tests before the implementation and write the code to fit to the tests. Although there is empirical indication that test-driven development increases quality [32, 186], having the developers create unit tests breaks the dual control principle. This might lead to missing defects on that level. How serious this problem is in comparison to the gains, however, is not known yet.

For unit tests, automation is important, as they should be run very often. For that, open source test frameworks, such as JUnit[12] for Java, RTR[13] for C# or Tessy from hitex for embedded systems, are available for many programming languages and environments. Ideally, unit tests check every change for consequences that break an existing test. This avoids that you perform large changes that break the system in various ways, because it then becomes difficult to separate all single issues and you need a high effort to resolve all of them.

In the unit test phase, the whole "driven by" dimension can be employed and you should use different test case derivation techniques to be most effective [116]. For example, for unit tests, it is straightforward to develop white-box tests to cover the code of the unit. To find the most defects, however, you should add black-box and other tests as well.

Most important is that you concentrate on an isolated unit. This allows you to test that unit in detail, to debug easier and to test early, even if the system is not complete yet. This comes at the cost, however, that you need to build drivers to run the tests and stubs which simulate the behaviour of missing units that you use in the unit under test. Moreover, when you test the units, you will need to add additional diagnosis interfaces to the units to make them easier to test. Usually, this additional effort is outweighed by the gains of thorough unit testing.

In general, in the quality control loop you should aim at having only unit test cases in the regression that show no failures. While developing changes to the system, unit tests will break, but the developer should realise that while the change is still not checked in. The rule should be that a developer should not check in code that breaks existing unit tests. Either the code or the tests need to be changed. To ensure this, you should include a unit test run after changes with the continuous build. In case the test run is very short, you could even test with each check-in.

If unit tests shows a failure, this gives a strong indication of the overall quality of the system. It is very likely that a user could experience a failure. Hence, a test showing a failure should never be accepted. In addition to the high probability of a user-visible failure, such tests in nightly quality analyses can build up over time, and with half the tests failing, nobody will take them seriously anymore.

To be able to test isolated units, we need a software architecture that specifies such units. This does not happen automatically when we build a software, but we need to consider it explicitly in the design of the system. We need to have an architecture of testable units.

Integration Test

While the developers create units and you test each unit by itself to assure its quality, you can start in parallel to integrate the finished units to larger subsystems. You can

[12] http://www.junit.org/

[13] http://rtr.codeplex.com

employ different integration strategies, such as bottom-up, top-down, big-bang or sandwich [171]. In all strategies, you replace stubs with real units, and in addition to the quality of the individual units, you are now interested in the quality of the interaction between units. Each unit could behave correctly to its specification and the intentions of the developers, but still their interaction could be wrong. These are interface defects. For example, consider a unit *Converter*, which converts data from another system and uses a unit *Separator* to break up a text file. *Converter* gives the text file to *Separator* and expects data chunks separated by semicolons. If the separation character was not explicitly specified, the developer of *Separator* could choose to use tabulators for separating the data chunks. Both units work correctly, but their interaction is wrong. In addition to correctness, integration testing is also interested in further properties of the interactions such as timing.

Massive parallel development of units and saving the cost for stubs seem to favour a big-bang integration. This means that you develop all units individually and then integrate them all at once. This holds the risk, however, that you get a large number of interface defects and you will struggle to sort them out. Therefore, I suggest to use incremental integration in which you add one or a small number of units after the other to build up the complete system (*incremental integration test*). This way, we can keep the number of integration problems at a manageable size. In fact, the best proven practice is continuous integration, i.e. a nightly build that integrates the current state of development and runs all unit and integration tests available so far. This combines perfectly with continuous quality control. In essence, a big-bang integration would bring you almost directly to a system test (see below) in which you continuously have to deal with simple interaction defects that still take a lot of time because the system test has to be restarted after fixing them.

Similar to unit tests, the aim should be to automate the integration tests as far as possible. Then we can include them in our continuous control loop and they help to identify interface defects in the interaction of units. In the incremental integration tests, we do not want to lose the integration we have reached so far. Therefore, we suggest to also run those integration tests that have at least once been successful in the hourly or daily build. If we employ levels of integration, from subsystem integration to system integration in an incremental integration, we have good support in localising defects.

System Test

After the integration test is finished, we a have a complete system. We can now test it against the specified functionality, performance and quality. We have all possibilities to mix tests driven by requirements, statistics and risks aimed at the whole system. You can start doing that in your test environment, but at some point, you should move the tests to the production environment at the customer. The final test by the customer is then the acceptance test.

Before the acceptance test, you can have user tests with actual (future) users of the system and you observe them while they interact with the system. This can have

different aspects to it. For example, a user can just be given the system without any further explanations to analyse how easy it is to learn the system. Without a detailed observation, user tests can already help to identify defects in the most commonly used parts of the system.

We want to stress that for system testing, the key to find the most defects is to use different techniques to cover all the aspects you are interested in. Below we will discuss the drivers of tests in more detail which should be used by system tests. Important is that you do not only concentrate on functional correctness but that you also go for performance or security at this point.

A difficulty with system tests is that not every test case is automatable. Therefore, you cannot run all system tests continuously. This would probably exceed your available effort by far. Nevertheless, you can still automate what can efficiently be automated and plan carefully when manual tests can give you the most information about the quality of the system. In particular, system tests give important input for the evaluation of the complete functionality and performance that are not measurable in earlier phases.

Drivers

We divide the forces that drive test case derivation into *requirements*, *structure*, *statistics* and *risk*. Requirements-driven tests only use the (functional) specification to design tests. Structure-driven testing, also called structural or white-box testing, relies on the source code and the specification. Statics-driven tests use statistical techniques to derive test cases. This can be completely random or driven by certain distributions that might also be constraint by what is sensible with respect to the interfaces and behaviour of the system. Risk-driven tests concentrate on covering the parts of the systems with the highest risks. This can be high probability of usage or failure as well as parts with the most sensitive information or the most dangerous behaviour.

Requirements

Requirements-driven tests only use the (functional) specification to design test cases. They are also called functional or black-box tests. Examples for functional tests are equivalence partitioning and boundary value analysis. In equivalence partitioning, the testers divide the input space into partitions that presumably trigger the same kind of functionality. Then they specify a test case for each partition. Boundary value analysis is complementary to this, as with it, the testers specify test cases for all boundary cases of the partitions.

A special case of requirements-driven testing is model-based testing where the requirements specification is substituted by an explicit and formal model. The model is significantly more abstract than the system so that it can be analysed more easily (e.g. by model checking). The model is then used to generate the test cases so that

the whole functionality is covered. For example, if the model is a state chart, state and transition coverage can be measured. If the model is rich enough, the test case generation and even the expected output can be generated automatically from the model. The model then also acts as oracle, i.e. it knows the correct output to an input. These benefits stand against the effort to build and maintain the model. With frequent requirements changes, it can be very efficient, because only the model needs to be changed but the test cases can be generated.

To drive tests by requirements includes the need that there are clear and complete requirements specifications. If they are not available, the tests themselves become the first formalisation of a requirement beyond code in the project. In small projects, this can be a valid and useful way. In larger projects, however, it is a very late point in the project to concretise requirements and it could lead to a lot of additional, unnecessary rework.

Requirements-driven tests are the basic part of tests in the quality control loop. We specify the requirements, functional requirements and quality requirements, from the quality model and check the conformance of the system with them. You should start as early as possible to formulate those requirements and reuse quality requirements and their operationalisation from other projects to save effort.

Structure

Structure-driven testing, also called structural, white-box or glass-box testing, relies on the source code and the specification. We divide structural testing into *control-flow* and *data-flow* techniques. In control-flow testing, the testers specify test cases according to the different possible paths of execution through the program. In contrast, they define test cases following the reading and writing of data in data-flow testing. There are measures to analyse the completeness of both kinds of structural tests. A set of test cases, a test suite, covers the control or data flow to a certain degree. For control-flow coverage, we measure the share of executed statements or conditions, for example. For data flow we measure the number and types of uses of variables.

Moreover, modern software systems come with a huge amount of parameters that can be set to configure the behaviour of the system. These settings are also called configurations. In practice, it is usually impossible to test all configurations because of the combinatorial complexity. Configuration testing is the method that generates test cases out of possible configurations in a structured way.

It is important to note that while structure-driven tests are important and necessary, because they give you a different point of view than requirements-driven tests, you should not overemphasise them. It is easy to measure a certain coverage of code by those tests. It is unclear, however, what that means for the quality of your system.

With the coverage by structure-driven tests, you only know that you have at least executed a statement or branch while testing. This is a basic indicator, but it is hard to interpret. If you only have a low coverage, it indicates bad quality because you

missed many parts of your system with your tests. If you have a high coverage, you still can have bad quality because there can be system parts missing or the covered parts do not follow the requirements correctly. Hence, you need to use more than just structure-driven tests, but they give you a basic indication of quality. Furthermore, in case your software system needs to conform to safety standards such as IEC 61508 or DO 178B, you are required to perform and document code coverage.

Statistics

Statistics-driven tests use statistical techniques to derive test cases. This can be completely random or driven by certain distributions that might also be constrained by what is sensible with respect to the interfaces and the behaviour of the system.

Random tests are generated without the specific guidance of a test case specification but use a stochastic approach to cover the input range. It does not rely on the specification or the internals of the system. It takes the interface of the system (or the unit if it is applied for unit testing) and assigns random values to the input parameters of the interface. The tester then observes whether the system crashes or hangs or what kind of output is generated.

Performance/load/stress tests check the behaviour of the system under heavy load conditions. They all put a certain load of data, interactions and/or users on the system by its test cases. In performance testing, usually the normal case is analysed, i.e. how long does a certain task take to execute under normal conditions. In load testing, we test similar to performance testing but with higher loads, i.e. a large volume of data in the database. Stress testing has the aim to stretch the boundaries of what is possible or expected load on the system. The system should be placed under stress with too many users interactions or too much data.

Risk

In risk-based testing, test cases are derived based on what in the system has a high risk. The methods assess risks in three areas:

- *Business or Operational.* High use of a subsystem, function or feature, criticality of a subsystem, function or feature, including the cost of failure
- *Technical.* Geographic distribution of development team, complexity of a subsystem or function
- *External.* Sponsor or executive preference, regulatory requirements

All of these areas can trigger specific test cases. For the business or operational risks, we employ, for example, testing based on operational profiles. They describe how the system is used in real operation. This usage then drives which test cases are executed. For example, imagine *manager* is a component in our system. Its subcomponents *input* and *output* are used differently. In 30 % of the usages, the *input* component, in 70 % the *output* component is executed. Our test suite follows

this with 70 % test cases that invoke *output* and 30 % test cases that call *input*. This way, tests driven by operational profiles act as a good representation of the actual usage.

The most common risk-based testing is probably *error guessing*, which belongs to the technical area. Experts familiar with the system and the domain predict which parts of the system and which situations and use cases are most likely to contain defects. You can then build test cases to cover these risks.

Penetration testing is an example for a test technique that considers criticality of a function. It derives test cases from misuse cases. Misuse cases describe unwanted or even malicious use of the system. These misuses are the basis for penetrating the system to find vulnerabilities in it. The result can be to either

- Crash the system,
- Get access to non-public data or
- Change data.

4.5.3 Effectiveness and Efficiency

Overall, testing is an effective quality assurance technique. On average, it is able to detect about half of the defects [200]. If you perform tests thoroughly and skip inspection before the tests, this value can go up to 89 %. If you perform tests poorly, however, the effectiveness can drop to 7 %. These values are similar over all major testing techniques.

One reason for the popularity of testing is probably that it is effective in many defect types. It is the most effective for data, interface and cosmetic defects [11]. Data defects are incorrect uses of data structures, interface defects are incorrect usage or assumptions on the interface of a component and cosmetic defects are spelling mistakes in error messages, for example. Structural tests are, in addition, effective for initialisation defects and control-flow defects. For many of the more specialised test techniques, there is no conclusive empirical knowledge.

The effort for finding a defect is also very similar over all test techniques. On average you need to spend 1.3 person-hours to find a defect with a range from 0.04 to 2.5. A defect, however, needs also to be removed. The test phase dimension differentiate here more clearly. It becomes the more expensive to remove a defect, the later it is removed [23, 200]. In unit testing, it takes 3.5 person-hours to remove a defect, in integration testing already 5.4 person-hours and in system testing 8.4 person-hours. These values are all higher than for automated static analysis or reviews and inspections.

4.5.4 Automation

For fast feedback to the developers in the quality control loop, we need to be able to run tests frequently. Manual tests are too expensive to execute on a daily basis.

Therefore, it is necessary to automate at least part of the tests so that they can be run often. Three parts of the test process can be automated: (1) generation of the test cases, test data and parameters; (2) execution of the test cases; and (3) evaluation of the test results in comparison with the expected results.

Generation

The generation of suitable test cases is intellectually the most difficult part in testing and therefore it is hard to automate. In practice, experienced testers and experts of the domain and system look for difficult and critical scenarios that are beneficial to test. In addition, they aim to cover the software – its code and requirements – with their tests so that they do not forget to test parts of the system. The goal is to test as complete as possible.

Especially these coverage criteria are the starting point of test case and test data generation. It is possible to generate test cases completely randomly, but this results in many invalid and useless test cases. In contrast, we can employ the code or models of the requirements to guide the test case generation. The stop criteria for the generation can be coverage of the code or the models.

In practice, generation of test cases is interesting, because a lot of effort is put into doing this manually. The effort that has to be spent for the automation, however, is also substantial. In our experience, it is most useful in situation in which vast amounts of test data or very long sequences of interactions are necessary to test a system. For example, a database-intensive business information system needs many business data or a software that implements a network protocol needs to be brought to its boundaries by creating long sequences of inputs and outputs.

Execution

In many software projects, the testers only automate test case execution. The challenges here lie in creating a test environment (system under test, stubs, hardware) in which tests can run completely automated as well as in maintaining the automated test cases. For certain programming languages and technologies, there is mature tool support for creating executing tests. For example, unit tests in Java can build on the JUnit framework which encapsulates many issues around executing and evaluating test cases. But also with a good framework, executable test suites can become very large and they need to be adapted to changes in the system. Therefore, they are a development artefact similar to the code that it tests. Hence, the same level of care and quality assurance is necessary for them. Then automating the test execution has a high potential to pay off.

Evaluation

Evaluating tests means that we compare what we got as a result from the tested system with the expected result. Depending on the type of system and output,

we can tolerate small deviations. For example, in real-time systems, we need to check whether messages arrive in a certain time frame. The expected results that we compare the actual results with can come from two sources: (1) the tester derived them from the specification or (2) an oracle. In many cases, while making the test cases executable, it is straightforward to implement the evaluation along with the test case execution. In some cases, however, calculating the correct result is difficult. For example, if the software under test does complex computations of aerodynamics, the tester cannot calculate that result by hand. Then an additional automated mean is needed – an oracle. For example, there might be simulation models of these aerodynamics which also produce a result that can be employed for the evaluation automation. If you used executable models for test case generation, they are also able to generate expected results.

4.5.5 Usage

Testing is essential in the quality control loop as it can directly show how a software system fails. If you work with our Quamoco quality models (Sect. 2.4), all activities in the model that involve end users need to be analysed by testing in addition to other quality assurance techniques, because testing can best resemble the actual user experience. Nevertheless, also testing has a limited defect detection capability and it tends to be more expensive than other techniques. Therefore, it should not be the sole method that you use in quality control. Only a combination can give you the optimal relation between effort and found defects.

The variety of testing techniques is large and each technique has different advantages and disadvantages. It also applies here that a combination is most advisable. What test techniques you should employ depends on the quality needs of your software system. As most experts consider functional suitability as the most important quality attribute, you should use functional tests that cover the specified requirements and additional expected properties. To improve defect detection, you can combine them with structural tests and measure code coverage.

For any non-trivial system, it is advisable to employ (automated) regression testing. The investment in these tests quickly amortises, because you save hours of debugging failures after changes to the system. Especially if you have regression tests on all levels of granularity, fault localising will be far cheaper. If your regression test suite becomes too large to be run completely after each change, you should at least run it in the nightly build and you can choose the most important test cases for each change.

If you need to use other test techniques depends largely on the type of system and its quality needs. In high-reliability or security environments, random testing is a good candidate to check the robustness. Reliability growth tests and penetration tests help in assuring these characteristics in a more focused way. In many cases, especially if there is a high risk, we propose to use performance tests.

4.5.6 Checklist

- Do you plan for and have an explicit architecture consisting of testable units?
- Have you planned for explicit unit, integration and system tests?
- If you plan to skip unit or integration tests, do you have a good reason for it?
- Do you employ a mixture of different test derivation techniques to maximise defect detection?
- Do you use other quality assurance techniques in combination with testing?
- Have you automated tests as far as possible to enable regression testing?

4.5.7 Further Readings

- Myers [161]
  The standard book on software testing. It has grown old a little, but it still contains the basics you need to know, described in an easy-to-read way.
- Beizer [15]
  A long-recognised book on software testing. It explains the whole process and many different techniques. It contains an especially detailed classification of defects.
- Spilner et al. [193]
  A book that focuses on what we left out in this section: test management. It gives a good overview and prepares you for the certified tester exam.
- Graham and Fewster [78]
  Although over 10 years old, this book still gives good insights into test automation.
- Broy et al. [28]
  One of the early comprehensive books on the available research in model-based testing.
- Beck [14]
  If you are interested in the idea of test-driven development and you want to learn more about how you can use it, this book is a good start. It is from one of the main founders of TDD and it is well written.

4.6 Product and Process Improvement

If applied successfully, continuous quality control allows to counter quality decay. Beyond that, it also provides additional insights that enable an organisation to *improve* quality and its *quality management* practice. This chapter discusses how the insights gained during product quality control can be used as input for improvements in the quality of processes, the quality models, the skills of the team and the overall

product. Hence, product quality control feeds information back into the overall continuous improvement for the PDCA cycle (see Sect. 4.1).

Quality control ensures that a product meets the defined quality requirements as deviations are detected and corrected. It does not automatically provide *quality improvement*, however; i.e. it does not raise an organisation's ability to produce high-quality products (the *manufacturing view* of Sect. 1.3). To provide this, you should translate the insights gained during quality control to incremental improvements of the development and quality assurance processes.

4.6.1 Lean Development

Process improvement comes down to changing the way people work. The work flows we practice in software development, however, are far too complex and variable to completely document them in a process description and then simply change that documentation to improve the process. Instead, we need the right mindset embraced by all employees. Everyone who participates in the development process needs to strive for excellence and thereby works for a high-quality product.

Such a mindset is what is behind many well-known approaches such as Total Quality Management (TQM), Six Sigma or ISO 9000. We found, however, that a lean development approach is the easiest to implement in practice and beneficial in most contexts. Especially the notion of *wastes* (originally *muda*) in the process characterises exactly what we look for in process improvement. Wastes are unnecessary steps in your process that do not add to the value (or the quality) of the product. In the Toyota Production System, from which lean development grew, we find these classes of wastes [154, 218]:

- Overproduction
- Waiting
- Handoff
- Relearning/reinvention
- Partially done work
- Task switching
- Defects
- Under-realising people's skills
- Knowledge loss
- Wishful thinking

All of them stem from problems they found at the Toyota production lines, but we can easily transfer them to software development [179]. Are we adding features the users do not need? Are we building components that are already available in libraries? Do we document our design decisions? All these questions lead to potential wastes in your process and all members of the development team need to feel responsible for looking for these wastes and to get rid of them.

Wastes are a vivid notion for what we should avoid. They are, however, only a part of lean development. The two central principles are *continuous improvement* and *respect for people*. We should all the time look for possibilities to advance our processes in a way that supports the people involved in these processes.

Continuous improvement includes recognising and removing wastes, but it also contains regular, so-called Kaizen events in which you identify small, incremental changes to the process. Implementing only these small changes improves the probability that you can succeed in changing the process, because modest changes in behaviour are far easier to make permanent. A single but large process change is likely to be refused by the team. The Kaizen event is a dedicated workshop with your team or a subset of your team to identify process improvements. It does not only consist of identifying wastes, but you can also decide to introduce additional process steps, for example, to increase cooperation with other teams or to improve quality standards. Holding it as a team meeting instead of letting the team leader decide also helps that the team will accept the process changes. In addition, continuous improvement denotes by *continuous* that there is no final process. There is always something to improve.

The respect for people constitutes an even more important part of process improvement in software development. Building or maintaining software is a task that extremely depends on the people that carry it out. Only highly motivated and supported team members can build high-quality software. This motivation and support need the corresponding respect.

How does that fit into continuous quality control and the control loop? On a higher level of abstraction, the mindset of lean development should give you a good basis for steering a project and developing your team. On a lower level, lean development's continuous improvement depends on the recognition of wastes that can be discussed at Kaizen events. The team members might recognise wastes while they perform the current process. But also the analysis of product quality in the quality control loop can give indications of wastes.

4.6.2 Derivation of Improvement Steps

In the quality control loop, the quality engineer gets feedback from the quality analysis on the current state of the product's quality. In this context, we need to distinguish internal quality analyses, which are done in-house, and external quality analyses, which collect field failures and customer satisfaction. Both kinds of analyses indicate that we need a better constructive quality assurance that prevents these quality defects. Results from external analyses, however, show in addition that we need to add more analytical quality assurance – internal quality analyses – so that the quality defects do not make it to the users.

In any case, for each quality defect reported from quality analyses, we need to analyse the root cause. Instead of only trying to get rid of the symptoms, we need to

Problem: Our software performs badly at a specific customer.

1. Why? Users wait minutes for queries to give results.
2. Why? The object-relational mapper needs too long for the mapping.
3. Why? The database system used by the customer was not tested.
4. Why? We concentrated on a standard configuration without communicating the compatibility list to our marketing.
5. Why? There is no interface between development and marketing that clarifies compatibility requirements.

Solution: Install explicit role in the process who is responsible for clarifying compatibility requirements.

Fig. 4.15 An example of the 5 Whys method

find and remove what caused the defect. In many cases, this is trivial. A developer misunderstood the requirements or forgot to implement a special case. For other problems, the root cause is very hard to find. Why does the software perform badly at a specific customer site?

A simple method from lean development can help here to get to the bottom of the problem. The 5 *Whys* want you to do what the name says: ask several times *why?* to understand cause and effects of a problem. Five is not a fixed number of questions but is a rule of thumb born out of experience. You can, however, also ask *why?* six or seven times. Let us look at the example in Fig. 4.15.

Having the root cause, the derivation of an improvement still tends to be complicated. If a developer misunderstood the requirements, the possibilities for improvement are endless. We could, for example, add more requirements engineers to the team so that they write more detailed requirements specifications. Or we change the way we specify requirements and add UML sequence diagrams and state machines. Which improvement fits best to your root cause depends on your context and we cannot give a clear guidance on that.

Nevertheless, it is a best practice to find small improvements that are incremental to the current process. You can, for example, find and discuss them in your regular Kaizen events. Revolutionary changes hold a high risk and should only be implemented in rare, extreme cases. Furthermore, it is desirable to have a measurable improvement so that we know when we have finished the improvement. You can feed this measurement back to the quality control process and thereby control when you reached your improvement goal.

4.6.3 Implementation of Improvement Steps

Implementing process improvements means changing the behaviour of people. This is only possible, if these people are also convinced of this change. How to implement change in an organisation has established itself as an area of research in its own

right [129, 130, 133]. We take a simple five-step method based on [181] from that research that helps you in implementing change. The five steps are:

- Convince stakeholders
- Create a guiding coalition
- Communicate change
- Establish short-term wins
- Make change permanent

Convince Stakeholders

We start by making a list of all stakeholders that are affected by an improvement measure. These stakeholders need to be convinced that the improvement is necessary and effective, even if there is a spirit of continuous improvement. We need the support of all people that are involved to make it a success.

We can achieve that by showing dramatic or surprisingly new information and data about quality defects, their effects on the users and the company, as well as the reasons for these quality defects. It is important that you touch the stakeholders on an emotional level to motivate them for change. Also keep in mind that different stakeholders will have different motives that you need to consider in convincing them. The test team might be interested in reducing certain defects so that they have less to report, the development team so that they have less rework.

Create a Guiding Coalition

It is easier to establish a process change, if not only one manager is responsible for it, but if there is a team that is the guiding coalition. This team should include people with certain characteristics: enough power to make decisions during the change implementation, broad expertise to make sensible decisions, credibility to explain the change, management to control the change and leadership to drive the change.

Communicate Change

The guiding coalition shows the goal behind the improvement to group leaders and developers. It can be frustrating for team members to change their routines without seeing what it is good for.

Establish Short-Term Wins

The best way to further motivate the people for keeping the process improvement is by giving them visible, measurable short-term wins. For example, by introducing a

static analysis tool, which the developers need to run before they check in code, we can build diagrams that show that it then reduces the defects found in inspections and unit tests and thereby the rework that the developers need to do.

Make Change Permanent

Short-term wins, however, do not realise the full potential of a process improvement. Therefore, we need to incorporate more and more employees into the change process. Furthermore, it helps to promote successful changes and to appoint new employees that help in using the new process. For example, a new employee could be responsible for selecting suitable static analysis tools, to keep them up to date and to integrate them into the standard development environment.

In the end, after you ensured that the change is permanent, you are left with the responsibility to check whether the improvement actually made something better. The control loop can help you to track whether the problems that brought you the process change persist or come up again. In that case, you need start from the beginning and ask more why questions to find the real root cause. If the problems are gone, congratulations! But remember that process improvement is continuous, because there is no perfect process.

4.6.4 Checklist

- Have you established continuous improvement and respect for people as key principles in your organisation?
- Do you check each quality analysis result for root causes and potential process improvements?
- Do you have regular Kaizen events to discuss wastes in your process and how you could remove them?
- Have you convinced all stakeholders of the process improvement?
- Have you created a guiding coalition?
- Have you communicated the change to all employees?
- Have you established short-term wins?
- Have you made the change permanent?

4.6.5 Further Readings

- Monden [154]
  The book on the initial Toyota production system. It is not directed at software development, but it gives you the historical background to lean development.

- Womack et al. [218]
  The story from Henry Ford to lean development written from an American perspective.
- Poppendieck and Poppendieck [179]
  Probably the earliest comprehensive application of the lean principles to software development.
- McFeeley [150]
  A handbook on software process improvement from the SEI. It consists of phases that you can map to our steps. These phases are, however, more deeply described.