PART I

Foundations

1 Software Quality

WHAT'S IN THIS CHAPTER?

- An overview of external and internal quality
- Discussions of technical debt and constructive quality assurance
- A look at various software metrics
- A brief look at tools for measuring and improving software quality

This book deals with software quality in PHP projects. What, exactly, do we mean by the term "software quality"? One example of a software quality model is FURPS (Functionality, Usability, Reliability, Performance, Supportability), which was developed by Hewlett-Packard.[1]

Although the FURPS quality model applies to all kinds of software, there are even more quality attributes with respect to Web applications, namely findability, accessibility, and legal conformity.[2] Software quality is a multifaceted topic, as Peter Liggesmeyer states in the introduction to *Software-Qualität: Testen, Analysieren und Verifizieren von Software, 2. Auflage*.[3]

[1]Robert Grady and Deborah Caswell, *Software Metrics: Establishing a Company-wide Program* (Prentice Hall, 1987. ISBN 978-0138218447).

[2]Klaus Franz, *Handbuch zum Testen von Web-Applikationen* (Springer, 2007. ISBN 978-3-540-24539-1).

[3]Peter Liggesmeyer, *Software-Qualität: Testen, Analysieren und Verifizieren von Software, 2. Auflage* (Spektrum Akademischer Verlag, 2009. ISBN 978-3-8274-2056-5).

> Every company developing software will attempt to deliver the best possible quality. But a goal can only be certifiably reached when it is clearly defined, which the term "best possible quality" is not. Software quality is multifaceted, thus software quality comprises many characteristics. Not all of these are equally important for the user and the manufacturer of the software.

A user's view on quality differs from a developer's view. We thus differentiate between *external* and *internal* quality, following Nigel Bevan's explanations of *ISO/IEC 9126-1*: *Software Engineering—Product quality—Part 1*: *Quality model* [4] in "Quality in use: Meeting user needs for quality."[5] In this chapter, we take a closer look at these two views.

EXTERNAL QUALITY

Customers, or the end users of an application, put their focus on quality aspects that are tangible for them. These quality aspects account for the *external quality* of the application.

- **Functionality** means that an application can actually fulfill the expected tasks.
- **Usability** means that a user can work efficiently, effectively, and satisfactorily with the application. Accessibility is a part of usability.
- **Reactivity** means short response times, which is crucial for an application in order to keep its users happy.
- **Security,** especially the security perceived by users, is another important factor for an application's success.
- **Availability** and **reliability** are especially important for Web applications with high user numbers. The applications must bear high loads and are required to work even in unusual situations.

All aspects of external quality can be verified by testing the application as a whole, using so-called *end-to-end tests*. The customer's requirements, for example, can be written down as acceptance tests. *Acceptance tests* not only improve the communication between the customer and the developers, but also make it possible to verify in an automated way that a software product fulfills all its functional requirements.

To improve an application's reactivity, we must measure the response time. We must use tools and techniques to find optimizations that promise the biggest win while keeping cost and effort low. To plan capacities, developers and administrators must identify potential future bottlenecks when an application is modified or traffic increases. All this information is required to assure the quality of an application with respect to availability and reliability in the long term.

[4]International Organization for Standardization, *ISO/IEC 9126-1*: *Software Engineering—Product quality—Part 1*: *Quality model*, *2008-07-29* (Geneva, Switzerland, 2008).

[5]Nigel Bevan, "Quality in use: Meeting user needs for quality," *Journal of Systems and Software* 49, Issue 1 (December 1999): 89–96, ISSN 0164-1212.

INTERNAL QUALITY

The needs of the developers and administrators of an application drive its *internal quality*. Developers put their focus on readable code that is easy to understand, adapt, and extend. If they do not do so, implementing the customer's future change requests becomes more difficult and thus more expensive over time. There is an increased danger that even small changes to the software will lead to unexpected side effects.

The internal quality of software is virtually imperceptible to customers and end users. End users expect software to satisfy all, or at least most, of their functional expectations and to be easy to use. If, upon acceptance, the product is "fast enough," most customers are satisfied.

Bad internal quality shows up in the longer term, though. It takes longer to fix even trivial bugs. Any changes or extensions to the software require a huge effort. Quite often, the developers sooner or later ask for a budget to clean up and refactor the code. Because customers or management often do not see the benefit of refactoring, these requests often are turned down.

Refactoring *means modifying the internal structure of software, without changing its visible behavior.*

Automated developer tests of individual software modules (unit tests), discussed in Chapter 2, allow for immediate feedback about new bugs that have been introduced when changing the code. Without automated tests, refactoring the code is a tough job.

A main goal of quality assurance, or to be exact, quality management, is to make the costs and benefits of internal quality transparent to all parties that are involved. Bad internal quality causes additional costs in the long term. If these costs can be quantified, it is possible to make the case for achieving good internal quality, because that reduces costs. This seems to be the only way of making management or the customer consider allocating a budget for code refactoring.

TECHNICAL DEBT

Ward Cunningham coined the term "technical debt":

> Although immature code may work fine and be completely acceptable to the customer, excess quantities will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product. Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering

> organizations can be brought to a standstill under the debt load of an unconsolidated implementation, object-oriented or otherwise.[6]

Cunningham compares bad code with a financial loan that has an interest rate. A loan can be a good idea if it helps the project to ship a product more quickly. If the loan is not paid back, however, by refactoring the code base and thus improving the internal quality, a considerable amount of additional cost in the form of interest piles up over time. At some point, the interest payments reduce the financial scope, until finally someone must declare bankruptcy. With regard to software development, this means that an application has become unmaintainable. Every small change to the code has become so expensive that it is not economically feasible to maintain the code.

Lack of internal quality tends to be more of a problem when development is being outsourced to a third party. Performing quality assurance, and especially writing unit tests, raises the development cost in the short term without an immediately measurable benefit. Because the focus often lies on reducing the project costs and keeping the time to market short, the developers have no opportunity to deliver high-quality code. The damage is done, however, and the customer must bear considerably higher maintenance costs in the long term.

It is crucial for every software project, and especially outsourced projects, not only to define quality criteria with regard to external quality, but also to ask for a sensible level of internal quality. Of course, this requires the customer to allocate a somewhat bigger budget, so the developers have some financial scope to account for internal quality.

Operating and maintenance costs of software are usually vastly underestimated. A medium-sized software project may last for one or two years, but the resulting application may be in operation for decades. The year 2000 problem proved that many applications are operational much longer than originally expected. Especially for applications that must be modified frequently, account for the biggest share of cost operation and maintenance. Web applications are known to require frequent changes, which is one of the reasons why many developers choose a dynamic language like PHP to implement them.

Other applications, for example, financial applications running on mainframes or telephone exchange software that needs to be highly available, are seldom modified. Although one new release per quarter may seem hectic for these kinds of applications, many Web applications require multiple releases each month.

Ron Jefferies reminds us that sacrificing internal quality to speed up development is a bad idea:

> If slacking on quality makes us go faster, it is clear evidence that there is room to improve our ability to deliver quality rapidly.[7]

It is obvious that the value of internal quality scales up with increasing change frequency of an application. Figure 1-1 shows that the relative cost of a bugfix in the coding phase of a project is 10 times, and in the operations phase is over 100 times, bigger than in the requirements phase. This proves that trying to postpone costs by delaying tasks in software development projects does not make sense from an economical point of view alone.

[6]Ward Cunningham, “The WyCash Portfolio Management System,” March 26, 1992, accessed April 17, 2010, `http://c2.com/doc/oopsla92.html`.

[7]Ron Jefferies, “Quality vs Speed? I Don’t Think So!” April 29, 2010, accessed May 1, 2010, `http://xprogramming.com/articles/quality/`.

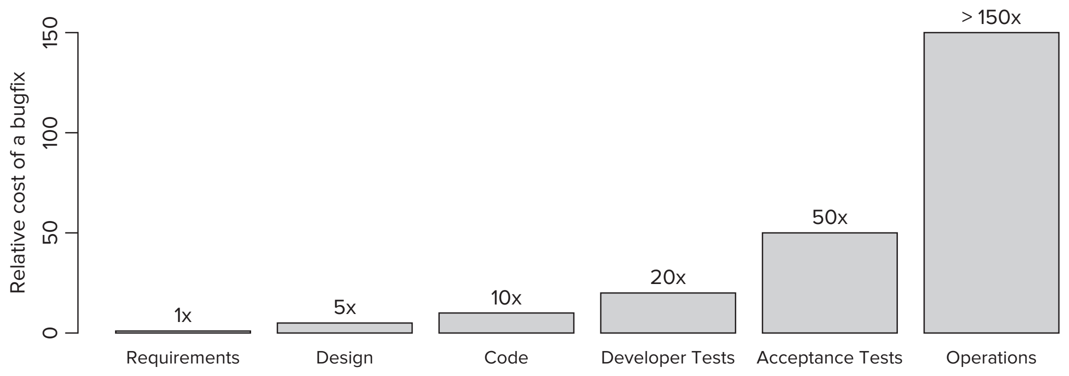

FIGURE 1-1: Relative cost of a bugfix[8]

CONSTRUCTIVE QUALITY ASSURANCE

Both *Capability Maturity Model Integration* (*CMMI*) and *Software Process Improvement and Capability Determination* (*SPICE*)[9] have a narrower view on quality assurance than many others, because they exclude testing.[10] All steps that CMMI and SPICE suggest with regard to organizational structure and process organization are prerequisites for the success of analytical activities like test and review of the finished software product and all measures of constructive quality assurance. Kurt Schneider defines constructive quality assurance as "measures that aim at improving selected software quality aspects on construction instead of afterward by verification and correction."[11]

The insight that avoiding bugs is better than finding and fixing them afterward is not new. Dijkstra wrote as early as 1972:

> Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover

[8]Barry Boehm, Ricardo Valerdi, and Eric Honour, "The ROI of Systems Engineering: Some Quantitative Results for Software-Intensive Systems," *Systems Engineering* 11, Issue 3 (August 2008): 221–234, ISSN 1098-1241.

[9]CMMI is explained in detail at `http://www.sei.cmu.edu/cmmi/`, and in ISO/IEC 12207. SPICE is covered in ISO/IEC 15504.

[10]Malte Foegen, Mareike Solbach und Claudia Raak, *Der Weg zur professionellen IT*: *Eine praktische Anleitung für das Management von Veränderungen mit CMMI*, *ITIL oder SPICE* (Springer, 2007. ISBN 978-3-540-72471-1).

[11]Kurt Schneider, *Abenteuer Softwarequalität—Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement* (dpunkt.verlag, 2007. ISBN 978-3-89864-472-3).

> that they should not waste their time debugging—they should not introduce bugs to start with.[12]

One approach to prevent the writing of defective software is *test-first programming*. Test-first programming is a technical practice that allows for constructive quality assurance by writing the test code before writing the production code. *Test-driven development*, which is based on test-first programming, ideally implies the following:

- All production code has been motivated by a test. This reduces the risk of writing unnecessary production code.
- All production code is covered by at least one test (code coverage). Modifications of the production code cannot lead to unexpected side effects.
- Production code is testable code and thus *clean code*.
- The pain that existing *bad code* causes is amplified, because that code cannot be tested or can be tested only with disproportional effort. This is a motivation to keep replacing existing bad code through refactoring.

Studies like that done by David S. Janzen[13] show that test-driven development can lead to significant improvements in developer productivity and better software quality.

Constructive quality assurance and normal software development cannot be clearly separated. Object-oriented programming and the use of design patterns improve the adaptability of software. Writing *clean code* (see next section) and concepts like a three-layer architecture or model-view-controller, when used properly, lead to significant improvements with regard to testability, maintainability, and reusability of the individual software components.

CLEAN CODE

In his book, *Clean Code*, Robert C. Martin lets Dave Thomas (among others) answer the question "what is clean code?":

> Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.[14]

[12]Edsger W. Dijkstra, "The humble programmer," *Communications of the ACM* 45, Issue 10 (October 1972): 859–866. ISSN 0001-0782.

[13]David S. Janzen, *Software Architecture Improvement through Test-Driven Development* (University of Kansas, Electrical Engineering and Computer Science, Lawrence, Kansas, USA, 2006).

[14]Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* (Prentice Hall International, 2008. ISBN 978-0-132-35088-4).

Steve Freeman and Nat Pryce add to this thought by stating that code that is easy to test must be good:

> For a class to be easy to unit-test, the class must have explicit dependencies that can easily be substituted and clear responsibilities that can easily be invoked and verified. In software-engineering terms, that means that the code must be loosely coupled and highly cohesive—in other words, well-designed.[15]

Let's take a closer look at these terms.

Explicit and Minimal Dependencies

All dependencies of a method to test must be clearly and explicitly defined in the method's API. This implies that all required objects must be passed either to the constructor of the class or to the tested method itself (*dependency injection*). Required objects should never be created in the method's body, because this disallows swapping them out for mock objects. The fewer dependencies a method has, the easier it becomes to write tests.

Clear Responsibilities

The *single responsibility principle* (*SRP*)[16] states that a class should have one clearly defined responsibility and should contain only those methods that are directly involved with fulfilling that responsibility. There should never be more than one reason to change a class. If the responsibility of a class is clearly defined and its methods are easy to call and can be verified through their return values, then writing unit tests for a class is a rather trivial task.

No Duplication

A class that does too much and has no clear responsibility is "a splendid breeding place for duplicated code, chaos, and death."[17] Duplicated code makes software maintenance more difficult, because each code duplicate must be kept consistent, and a defect that has been found in duplicated code cannot be fixed in just one spot.

Short Methods with Few Execution Branches

The longer a method is, the harder it is to understand. A short method is not only easier to understand and reuse, but also easier to test. Fewer execution paths means that fewer tests are required.

[15]Steve Freeman and Nat Pryce, *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley, 2009. ISBN 978-0-321-50362-6).

[16]Robert C. Martin, *Agile Software Development. Principles, Patterns, and Practices* (Prentice Hall International, 2002. ISBN 978-0-135-97444-5).

[17]Martin Fowler, *Refactoring. Wie Sie das Design vorhandener Software verbessern* (Addison-Wesley, 2000. ISBN 3-8273-1630-8).

SOFTWARE METRICS

There are various software metrics for measuring internal quality. They are the basis for quantifying the costs that emerge from bad internal quality.

> *A software metric is, in general, a function that maps a software unit onto a numeric value. This value says how well a software unit fulfills a quality goal.*[18]

Testability is an important criterion for maintainability in the ISO/IEC 9126-1 software quality model. Examples for quantifying the testability based on object-oriented software metrics can be found in "Predicting Class Testability using Object-Oriented Metrics" by Magiel Bruntink and Arie van Deursen,[19] and in "Metric Based Testability Model for Object Oriented Design (MTMOOD)" by R. A. Khan and K. Mustafa.[20]

A good overview of object-oriented software metrics is *Object-Oriented Metrics in Practice*: *Using Software Metrics to Characterize*, *Evaluate*, *and Improve the Design of Object-Oriented Systems* by Michele Lanza and Radu Marinescu (Springer, 2006. ISBN 978-3-540-24429-5).

The following sections discuss some metrics that are especially relevant for testability.

Cyclomatic Complexity and npath Complexity

The *cyclomatic complexity* is the number of possible decision paths in a program or program unit, usually a method or class.[21] It is calculated by counting the control structures and Boolean operators in a program unit, and it represents the structural complexity of a program unit. McCabe claims that a sequence of commands is easier to understand than a branch in the control flow.

A large cyclomatic complexity indicates that a program unit is susceptible to defects and hard to test. The more execution paths a program unit has, the more tests are required. The npath complexity counts the number of acyclic execution paths.[22] To keep this number finite and eliminate redundant information, the npath complexity does not take every possible iteration of loops into account.

[18]Schneider, *Abenteuer.*

[19]Magiel Bruntink and Arie van Deursen, "Predicting Class Testability using Object-Oriented Metrics," *SCAM '04*: *Proceedings of the Source Code Analysis and Manipulation*, *Fourth IEEE International Workshop* (2004): 136–145. ISBN 0-7695-2144-4.

[20]R. A. Khan and K. Mustafa, "Metric Based Testability Model for Object Oriented Design (MTMOOD)," *SIGSOFT Software Engineering Notes* 34, Issue 2 (March 2009): 1–6. ISSN 0163-5948.

[21]Thomas J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering* 2, No. 4 (IEEE Computer Society Press, Los Alamitos, CA, USA, 1976).

[22]Brian A. Nejmeh, "NPATH: A Measure of Execution Path Complexity and its Applications," *Communications of the ACM* 31, Issue 2 (February 1988): 188–200. ISSN 0001-0782.

Change Risk Anti-Patterns (CRAP) Index

The *Change Risk Anti-Patterns (CRAP) Index*, formerly known as *Change Risk Analysis and Predictions Index*, does not directly refer to testability. We mention it here because it is calculated from the cyclomatic complexity and the *code coverage* that is achieved by the tests.

Code that is not too complex and has adequate test coverage has a low CRAP index. This means that the risk that changes to the code will lead to unexpected side effects is lower than for code that has a high CRAP index. Code with a high CRAP index is complex and has few or even no tests.

The CRAP index can be lowered by writing tests or by refactoring the code. The refactoring patterns *extract method* and *replace conditional by polymorphism*, for example, allow for shortening methods and reducing the number of decision paths, and thus the cyclomatic complexity.

Non-Mockable Total Recursive Cyclomatic Complexity

Miško Hevery, creator of the so-called Testability Explorer (`http://code.google.com/p/testability-explorer/`), a tool to measure testability of Java code, defined the *non-mockable total recursive cyclomatic complexity* software metric. The name is composed of the following parts:

- **Cyclomatic complexity:** This is the structural complexity of a method.
- **Recursive:** We look at the cyclomatic complexity of a method and take into account the cyclomatic complexity of the called code.
- **Total:** The structural complexity of object creation is also taken into account.
- **Non-mockable:** Any dependent code that can be replaced by a mock object is ignored. A mock object replaces the real object for testing purposes (see Chapter 2).

Basically, the *non-mockable total recursive cyclomatic complexity* measures the amount of complex code that cannot be replaced by mock objects for unit testing purposes. These kinds of complex dependencies that disallow isolating code for testing purposes lead to "pain" when testing. All these dependencies should be refactored, for example by introducing *dependency injection*, so that they can be replaced by mock objects.

Global Mutable State

The *global mutable state* is another metric that Miško Hevery has defined for his Testability Explorer. It counts all elements of the *global state* that a program unit writes to or could possibly write to. In PHP, these are all global and superglobal variables and static class attributes.

Changes to the global state are a side effect that not only makes each test more complex, but requires that every other test be isolated from it. PHPUnit, for example, supports saving and restoring the global and superglobal variables and static class attributes prior to running a test, and restoring them after the test, so that modifications of the global state do not make other tests fail. This isolation, which can be further enhanced by executing each test in its own PHP process, is resource-intensive and should be avoided by not relying on a global state.

Cohesion and Coupling

A system with strong cohesion is comprised of components responsible for exactly one clearly defined task. Loose coupling is achieved when classes are independent from each other and communicate only through well-defined interfaces.[23] The Law of Demeter[24] requires that each method of an object calls methods only in the same object and methods in objects that were passed to the method as parameters. Obeying the Law of Demeter leads to clear dependencies and loose coupling. This makes it possible to replace dependencies by mock objects, which makes writing tests much easier.

TOOLS

Software quality is multifaceted, and equally multifaceted are the tools that PHP developers can use to measure and improve the software quality of PHP projects.

PHPUnit

PHPUnit (`http://phpun.it/`) is the de-facto standard for unit testing in PHP. The framework supports writing, organizing, and executing tests. When writing tests, developers can make use of the following:

- Mock objects (see Chapters 2 and 9).
- Functionality for testing database interaction (see Chapter 10).
- An integration with Selenium (see Chapter 11) for browser-based end-to-end tests. Test results can be logged in JUnit and code coverage as Clover XML for continuous integration purposes (see Chapter 12).

phploc

phploc (`http://github/sebastianbergmann/phploc`) measures the scope of a PHP project by means of different forms of the *lines of code* (*LOC*) software metric. In addition, the number of namespaces, classes, methods, and functions of a project are counted, and some values, like the average complexity and length of classes and methods, are counted. Chapter 12 shows an example of how phploc can be used.

PHP Copy-Paste-Detector (phpcpd)

The PHP Copy-Paste-Detector (phpcpd) (`http://github/sebastianbergmann/phpcpd`) searches for duplicated code, the so-called *code clones* in a PHP project. Chapter 12 shows how phpcpd can be used for an automated and regular search for duplicated code in the context of continuous integration.

[23]Edward Yourdon and Larry Constantine, *Structured Design*: *Fundamentals of a Discipline of Computer Program and Systems Design* (Prentice Hall, 1979. ISBN 978-0138544713.)

[24]K. J. Lienberherr, "Formulations and Benefits of the Law of Demeter," *ACM SIGPLAN Notices* 24, Issue 3 (March 1989): 67–78. ISSN 0362-1340.

PHP Dead Code Detector (phpdcd)

The PHP Dead Code Detector (phpdcd) (`http://github.com/sebastianbergmann/phpdcd`) searches PHP projects for code that is not called anymore and thus potentially can be deleted.

PHP_Depend (pdepend)

PHP_Depend (pdepend) (`http://pdepend.org/`) is a tool for static code analysis of PHP code. It is inspired by JDepend and calculates various software metrics, for example the cyclomatic complexity and npath complexity that were previously mentioned. It also is possible to visualize various aspects of software quality. Chapter 12 shows how PHP_Depend can be used in the context of continuous integration, to keep an eye on relevant software metrics while developing.

PHP Mess Detector (phpmd)

The PHP Mess Detector (phpmd) (`http://phpmd.org/`) is based on PHP_Depend and allows the definition of rules that operate on the "raw data" software metrics that PHP_Depend has calculated. If a rule is violated, for example because the cyclomatic complexity exceeds a given limit, a warning or an error is triggered. Chapter 12 shows how the PHP Mess Detector can be used in the context of continuous integration.

PHP_CodeSniffer (phpcs)

The PHP_CodeSniffer (phpcs) (`http://pear.php.net/php_codesniffer/`) is the most commonly used tool for static analysis of PHP code. Its countless sniffs to detect *code smells*[25] range from formatting rules via software metrics to the detection of potential defects and performance problems. Chapter 12 shows how PHP_CodeSniffer can be used in continuous integration to enforce a certain coding standard.

bytekit-cli

bytekit-cli (`http://github.com/sebastianbergmann/bytekit-cli`) is a command line front-end for the Bytekit PHP extension (`http://bytekit.org/`). Bytekit allows for code introspection at bytecode level. With bytekit-cli it is possible to find code that generates output for a code review. Disassembling and visualizing of PHP bytecode is also possible.

PHP_CodeBrowser (phpcb)

The PHP_CodeBrowser (phpcb) (`http://github.com/mayflowergmbh/PHP_CodeBrowser`) is a report generator taking the XML output of other tools like the PHP Copy-Paste-Detector, PHP_CodeSniffer, and PHP Mess Detector as input. It generates a unified report, which is extremely useful in continuous integration (see Chapter 12).

CruiseControl and phpUnderControl

phpUnderControl (`http://phpUnderControl.org/`) is a modification and extension of CruiseControl (`http://cruisecontrol.sourceforge.net/`), the Java open-source solution that originally made continuous integration popular. Sebastian Nohn, in 2006, was one of the first to use

[25]Fowler, *Refactoring.*

CruiseControl in PHP projects.[26] In a meeting of the PHP Usergroup Dortmund in Germany, which was attended by Manuel Pichler, Kore Nordmann, and Tobias Schlitt, the idea was born to simplify the configuration of a continuous integration environment for PHP projects based on CruiseControl. The result was phpUnderControl, which—like CruiseControl in the Java world—has made continuous integration popular in the PHP world. Manuel Pichler and Sebastian Nohn describe how to install, configure, and operate phpUnderControl in Chapter 12.

Hudson

Like CruiseControl, Hudson (`http://hudson-ci.org/`) is an open-source solution for continuous integration. In the Java world, Hudson is superseding the outdated CruiseControl. This is not surprising, because Hudson is more robust and easier to handle, and it is being actively developed. The php-hudson-template project (`http://github.com/sebastianbergmann/php-hudson-template`) is a configuration template for PHP projects in Hudson.

Arbit

Arbit (`http://arbitracker.org/`) is a modular solution for project management. It features an issue tracker, a wiki, a code browser, and a continuous integration server. Arbit is currently still in alpha state and thus not really suited for production use. You should keep an eye on the project though.

CONCLUSION

A software quality goal can only be reached when it has been defined precisely. The software metrics that have been introduced in this chapter can help to define these goals. Instead of just gathering data because the continuous integration server makes it possible, the data should be used to answer dedicated questions about the quality of the inspected software product. The *Goal-Question-Metric* (*GQM*) approach by Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach,[27] summarized by Kurt Schneider in just one sentence, can help:

> Do not measure what is easy to measure, but what you need to reach your improvement goals.[28]

This chapter outlined some goals for improving the internal quality of software, for example testability, maintainability, and reusability. We introduced some software metrics to measure these aspects. We hope the discussion of "technical debt" will improve mutual understanding between the various parties involved in a software project and made clear the importance of internal quality of software.

Most Web applications are changed and adapted quite frequently and quickly. Their environment, for example the size and behavior of the user base, is constantly changing. The internal and external quality are just snapshots. What was sufficient yesterday can be insufficient today. In a Web environment, it is especially important to monitor and continuously improve the internal quality, not only when developing, but also when maintaining the software.

[26]Sebastian Nohn, "Continuous Builds with CruiseControl, Ant and PHPUnit," March 7, 2006, accessed April 28, 2010, `http://nohn.net/blog/view/id/ cruisecontrol_ant_and_phpunit`.

[27]Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach, "Goal Question Metric Paradigm," *Encyclopedia of Software Engineering,* 2 Volume Set (John Wiley & Sons, 1994. ISBN 1-54004-8.)

[28]Schneider, *Abenteuer.*