Conference Journal 2017

Interesting Insights into Professional Practice

Papers of Lecturers at the Software Quality Days

2017 software quality days

EXPERIENCE THE VALUE OF QUALITY

Impressum

Publisher / Herausgeber

Software Quality Lab GmbH
Gewerbepark Urfahr 6
4040 Linz
Austria

www.software-quality-lab.com
info@software-quality-lab.com
+43 5 0657-0

Published by Software Quality Lab GmbH, January 2017

2018 software Quality days

EXPERIENCE THE VALUE OF QUALITY

16th – 19th January, 2018 Vienna

SUBMIT YOUR PAPERS FOR THE SOFTWARE QUALITY DAYS!

You are kindly requested to submit your application for a presentation, tutorial or workshop to be included as part of the conference.

↗ **Submit your application now!** www.software-quality-days.com

ONLY UNTIL MAY 31

WE LOOK FORWARD TO WELCOMING YOU IN 2018!

Key topic

Software Quality 4.0: Advanced Methods and Tools for better Software and Systems

Topics

Requirements, testing, automation, agile, project management, quality management, processes, techniques and methods

- Practical tracks
- Scientific track
- Solution Provider Forum
- Top keynote speakers
- Lectures
- Workshops and tutorials

Highlights

TOOL CHALLENGE

BEST QUALITY TOOL AWARD

CELEBRATE 10 YEARS SOFTWARE QUALITY DAYS

↗ **Submit your application now!**
www.software-quality-days.com

Content

Continuous IoT Testing using IaaS

Towards flexibility and scalability in automated IoT testing using on-demand infrastructure

Test automation evolves – its requirements change based on the system under test as well as organizational demands. On the one hand, the increasingly agile world of software development strives for continuous testing to provide immediate feedback on code changes. On the other hand, the Internet of Things targets the interconnection of billions of devices, generating lots of data and a respective load on central IoT components. How can test automation be available on demand, while scaling with ever-growing demands on test infrastructure and tooling of the IoT? This report sketches our approach.

1. The Value of Test Automation

In the increasingly agile world of software development, the automation and repetitive execution of tests is part of the everyday business. Developers commit their changes, continuous integration systems build the updated software artifacts, and trigger the respectively automated tests afterwards. If all tests pass, the new software build is declared as successful and the artifacts are deployed to execution environments.

Taking this generic model of the continuous integration process into account, the value of test automation might be paraphrased as the percentage of time the development team does not actively use for testing and is, thereby, focused on the product's development.

While, in the past, test automation focused on the automated execution of selected tools to test a well-known and long-established software under test, emerging technologies like microservices and the Internet of Things (IoT) impose a much broader scope on test automation, requiring a high level of flexibility with respect to tools and environments.

Flexibility, which we consider as another aspect of test automation's value, can be supported by intense reuse. However, limiting this reuse to tools and scripts falls short, as also test infrastructure should be dynamically reused. This is especially essential, as there is usually more than one development team, and each of these teams faces the same challenges for efficient test automation.

Further, and mostly because the IoT is about connecting billions of devices, it is no longer sufficient to have single test machines running a single tool. It is, instead, necessary to scale the test infrastructure on demand, according to the requirements of the system under test.

Within this article, we would like to share our approach to and experience with the automation of IoT tests. We demonstrate how we use infrastructure-as-a-service providers and generic configuration tools, to provide a flexible and scalable solution to continuous testing, with a focus on IoT environments.

The next section provides a brief overview on the history of our test automation efforts and their evolution over time. Section 3 will outline the challenges of IoT testing in detail and thereby formulate requirements for their solution. After a short investigation of available tools in section 4, we will describe our approach in section 5. Eventually, we summarize our gained experience and provide an outlook on the planned next steps.

2. Our History of Test Automation

When we started the creation of a framework for automated system regression tests at the end of the last decade, the system under test was an in-house developed, monolithic client/server-architecture for enterprise application integration (EAI) and business process management (BPM).

In a first step, test automation focused on the server component of the system. Tests should provide confidence in the functionality of the BPM execution engine, as well as in the connectors to external systems. For this chosen approach we faced advantages as well as disadvantages:

Implementation largely benefits from the "eating your own dog food" approach, where the BPM execution engine is both: SUT and technical basis for the framework.

Large number of required test systems (> 40 virtual & physical machines) renders moving to a new server environment virtually impossible.

With the emergence of more automation tools, the number of separate reports increased, making it more cumbersome to get the "full picture" of test status.

About three years ago, the former system under test was integrated into a so called “software suite”, denoting an early prototype of a platform-as-a-service offering. Now the EAI system was only one component of multiple server-based applications with (more or less) freely configurable interactions.

These mere technical transitions were accompanied by organizational changes:

Stakeholders for each platform component are spread over several company locations and require aggregated test reports for decision making.

New demand for automated functional and non-functional tests of each component (in addition to platform integration tests)

New requirement: Changing the test environment on-demand with almost no overhead

Based on our skills and experience, we again chose to use the EAI component as a test driver, which now also integrated external test tools like JMeter and SoapUI.

Despite this flexibility of our new automation framework, we observed drawbacks of this approach, caused by much greater amount of data to be processed and the constantly increasing organizational demands, which limited the frequency of test runs against changing target environments.

With the turn of the software suite into a cloud-based IoT platform offering about a year ago, in combination with our organizational transformation from a “test execution department” into a “test services provider”, we need to react more quickly and flexible to our customers’ requests.

3. Challenges in IoT Testing

The subject of the Internet of Things, and established trends in modern software development, impose various demands on software testing; technical and organizational as well.

As stated above, software development teams strive for focusing their efforts on the product development, while ensuring an appropriate level of quality by putting – ideally – only minimal efforts into testing. Nevertheless, in general, functional tests are managed by the team itself during the day-to-day work, e.g., by unit testing or small automated system tests.

For non-functional tests, such as performance tests, teams seldom have the required resources, neither human, nor in terms of available test machines. Thus, such tests are postponed to late pre-production test phases, or worse, put in second place, which is obviously risky, especially for IoT components that must be designed to handle billions of connected devices.

> Challenge 1: Teams must be enabled to integrate non-functional tests into their continuous tool chain, without requiring them to manage the needed test environments.

Even more versatile, than the need for test infrastructure, is the list of tools that are used for testing. There might be a common consensus, like using JMeter for load testing and Selenium for testing web applications. But, as the central object in IoT is the “thing”, tests must also be able to integrate things, either physically or virtually.

> Challenge 2: The tools used for testing a specific software artifact cannot be foreseen. Thus, integrating new tools, such as device simulators, must be easy.

Although the platforms forming the Internet of Things are, by definition, globally available, and though agile development processes proclaim shippable product increments, testing new software artifacts is often done in more private networks. However, switching between test environments should only require minimal effort.

> Challenge 3: Test environments and systems under test are not static. Teams must be enabled to test their system in and from different networks and from varying geographic locations.

To summarize the above: An efficient support for continuous testing in IoT environments must

be available on demand,

be flexible with respect to tools, environments, and locations, and

scale with increasing demands.

4. Short Market Analysis

When searching the market for publicly available tools that fit our needs, we find basically two categories of approaches:

tool specific services, where the scalable usage of a concrete tool, e.g., JMeter, is provided via API and/or a web-based GUI.

generic infrastructure services, where virtual machines can be dynamically acquired with almost arbitrary dimensions as regards CPU and memory, as well as different geographic locations.

The tool specific services are highly sophisticated with respect to their tool’s domain. Thus, if you know how to use the tool, there is almost no ramp-up until you can use the service itself. Most of these services work with a pay-as-you-go or monthly plans so costs are quite low, even for large scale scenarios.

However, as we require more than one specific tool, we would need to use multiple of these public offers, each of them having its own API, payment model, and handling, resulting in a lot of overhead. Furthermore, finding a suitable public service for a specific tool might be tedious, if not impossible.

In contrast, Infrastructure-as-a-Service (IaaS) providers, such as Amazon EC2 or Microsoft Azure, are much more generic by simply providing the ability to create almost arbitrarily configured virtual machines. They also rely on pay-as-you-go, invoicing only the resources and time you acquire. Unfortunately, there are no generic interfaces for installing software on the created VMs. Thus, though providing high flexibility with respect to VM sizing and geographic locations, there is no default way to automatically run tools on this infrastructure.

In addition to their respective pros and cons, both service categories only work for systems under test that can be reached on the internet. As stated above, we also need to support internal environments, which is not addressed by any of these.

5. Achieving Scalability and Flexibility

Despite their lack of installing software, IaaS providers supply a very flexible and scalable basis for solving our IoT test challenges. Fortunately, there exist several tools for automating software roll-out and system configuration. Such a (software) configuration management tool, therefore, is the perfect complement to the dynamic infrastructure allocation provided by IaaS.

From the set of available tools, we chose Chef, for the following reasons:

Chef provides a local execution mechanism sparing us the necessity of connecting every machine to a central server.

Chef is open source and can be used at no charge.

There is a community providing ready-to-use Chef routines, named cookbooks in the Chef-universe, for a lot of installation and configuration tasks.

Using Chef's local mode, we are not restricted to configuring the machines allocated online from IaaS providers, but are enabled to use the exact same mechanism on machines available in local networks.

Combining automated infrastructure management, on the one hand, with the automated configuration of this infrastructure, on the other hand, we can formulate a general schema for executing an almost arbitrary test scenario.

Setup Phase

Allocate the required number of test machines. This includes the specification of system parameters, such as number of CPU cores.

Execution Phase

Install and configure the respective tools or scripts on the previously allocated machines, run the test, and save report artifacts, e.g. log files or screen shots.

Teardown Phase

Deallocate the test machines. This can be a deletion of virtual infrastructure, as well as a simple release of machines.

From this generic flow, we can derive the layered model depicted in figure 1, showing the architecture of our solution. On top of the virtual or physical infrastructure we create two layers of abstraction. The first one, shown on the left, provides a unified interface for allocating and releasing machines. The second one offers a generic entry point for configuring these machines. By meaningful combination of these two interfaces we are enabled to run fully automated test scenarios.

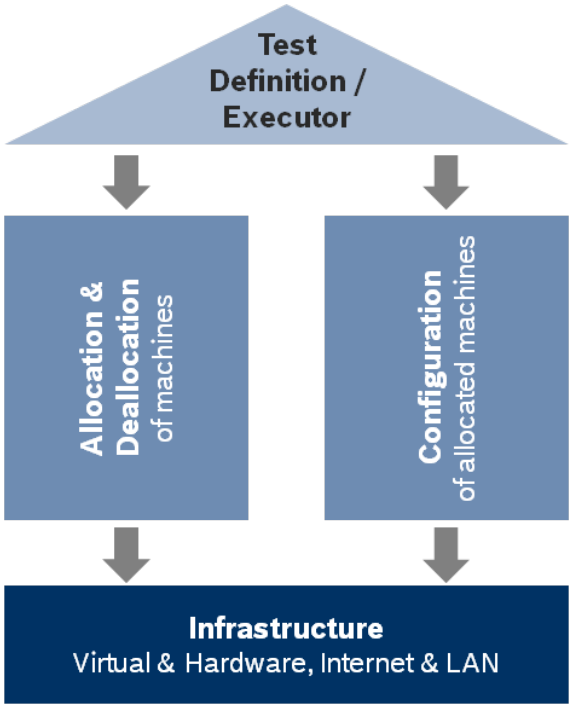


Figure 1 - General Solution Architecture

Using IaaS providers for the setup phase, we can scale with the test scenario – even in an on-demand manner. For small scenarios, a single machine with four CPU cores and the same number of RAM gigabytes may be sufficient, whereas for high load scenarios, we might require ten 20-core machines with respective memory sizing. In summary: No matter the size of the scenario, we have the required infrastructure at hand.

Relying on Chef for the execution phase enables the flexible roll-out of software - comprising test tools and systems under test - as long as this configuration

can be specified in terms of a cookbook. Thus, integrating new tools is simple – we just need to create a new cookbook or adapt an existing one, solving the issue raised by challenge 2.

As a side effect of using IaaS providers like Amazon EC2 we can allocate machines all over the world, based on the locations of the provider's data centers. Further, and since our approach abstracts from a concrete IaaS provider, we can consider an entity managing virtual or physical machines inside a local network as another infrastructure provider. The combination of these two aspects leads to the solution of challenge 3.

Covering challenge 1 (providing easy-to-use and ready-to-go solutions for teams) is the final step of our concept. As of this writing, this step is still in progress. Nevertheless, creating the generic interfaces for automating the basic steps of a test scenario has been a major milestone on that path.

6. Summary and Outlook

Since starting our test automation journey some years ago, the systems under test continuously changed, and with them the organization and our skills evolved.

Today, at the dawn of the Internet of Things era, we face new challenges in automated testing, requiring a high degree of scalability and flexibility with respect to test environments and tools. Further, such automation must be easy to use and available on demand.

In this article, we have provided insights on our solution towards flexible and scalable IoT test automation. This approach combines Infrastructure-as-a-Service techniques and the configuration management tool Chef into an abstract layer for providing test environments. By the creation of generic interfaces to this layer, we enabled the execution of arbitrary scenarios, giving us the ability to create automated test services tailored to our customers' needs.

In order to provide these services faster for customers in an agile IoT development world, several concepts are conceivable:

We can provide pre-configured test infrastructure packages (load & performance, Selenium WebDriver, Jenkins, testcase management) upon request, or establish a self-service via REST API for creating instant test environments on demand. The latter represents a major step towards integrating automated IoT testing into the continuous tool chain.

Being part of the Bosch IoT cloud services, this could also be offered as a charged service for external customers who need to test their own products within real-world conditions.

We know that there are further challenges ahead, as we still dive deeper into the world of things. But using the presented approach as a basis, we feel well-prepared to accept and solve them.

Authors

Felix Elliger

System engineer at Bosch Software Innovations GmbH.

After starting as a research and software developer in 2009 he moved to QA in 2012 having a strong focus on test automation.

Felix.Elliger@bosch-si.com

Dr. Jürgen Kraus

System engineer at Bosch Software Innovations GmbH.

He moved from software development to QA in 2009 with test automation and tooling as key subjects.

Juergen.Kraus@bosch-si.com

Co-author

Alexander Rothenberg

Senior system engineer at Bosch Software Innovations GmbH.

Started as a functional tester in 2007 and since this he moved his focus to test automation frameworks and performance testing.

Alexander.Rothenberg@bosch-si.com

Werkzeuggestützte Sicherheitszertifizierung

Anwendung auf den Industrial Data Space

Eine steigende Verbreitung von Netzwerk-basierten Produkten führt zu einem starken Anstieg von potenziellen Angriffszielen. Für viele dieser Produkte ist nicht abschätzbar, wie es um deren Sicherheit bestellt ist. Diesem Umstand kann durch eine effizientere Sicherheitszertifizierung und vor allem Rezertifizierung bei Änderungen entgegen getreten werden. In dieser Arbeit wird mit CARiSMA ein Sicherheitsmodellierungs- und Analysewerkzeug vorgestellt, welches notwendige Ansätze zur automatisierten Zertifizierung und Rezertifizierung bietet, sowie das Konzept eines automatisierten (Re-)Zertifi-zierungsprozesses basierend auf CARiSMA.

Da Computer immer kleiner und günstiger werden nimmt deren Verbreitung in unserem Alltag sowohl auf der Arbeit als auch zu Hause zu. Allgegenwärtige Geräte, wie Smartcards, RFID-Tags, Sensornetzwerke und persönliche digitale Geräte erfüllen vielfältige hilfreiche Aufgaben, speichern jedoch für diese Aufgaben auch viele persönliche Daten. Aus diesem Grund sind diese Geräte potenzielle Angriffsziele und besonders schützenswert. Umfassende Ansätze zur Absicherung solcher Geräte und Netzwerke existieren bisher kaum.

Eine Umfrage im Rahmen des BITKOM Leitfadens Cloud-Computing von 2009 nennt die Sicherheit von Cloud-Computing-Diensten als eines der wichtigsten Hindernisse bei der Akzeptanz von Cloud-Computing-Systemen in Unternehmen. Die Ursache ist zum einen im mangelnden Einsatz und Unterstützung von Sicherheitstechnik und zum anderen in signifikanten Anforderungen an Skalierbarkeit und Elastizität der Cloud-Computing-Systeme zu sehen, für die derzeit erst spezielle Sicherheitstechniken entwickelt werden. Die Sicherheitsthematik im Cloud-Computing-Bereich hat daher zur Gründung von Nutzerverbänden, z.B. der Cloud Security Alliance geführt, welche Unternehmen bei der Sicherheitsevaluierung von Cloud-Computing-Systemen unterstützt.

Für die Förderung und Zertifizierung von Datenaustausch und die Nutzung gemeinsamer Systeme zwischen verschiedenen Unternehmen im Rahmen von Industrie 4.0 wurde analog zu den Nutzerverbänden im Bereich des Cloud-Computing Anfang 2016 der Industrial Data Space e.V. gegründet [1].

Eine Softwarekomponente, welche in der Cloud oder dem Industrial Data Space eingesetzt wird, muss entsprechend der Richtlinien der jeweiligen Verbände sowie nach gesetzlich festgelegten Standards zertifiziert werden. Dies ist ein aufwändiger und fehleranfälliger Prozess, der für jede Änderung an der Softwarekomponente erneut durchgeführt werden muss.

Unser Sicherheitsmodellierungs- und Analysewerkzeug CARiSMA bietet Ansätze, die in einer teilautomatisierten und effizienten Erstzertifizierung angewendet werden können und auch bereits exemplarisch im Rahmen des Cloud-Computing angewendet wurden. Basierend auf diesen Ansätzen werden in dieser Arbeit die Anforderungen und Abläufe einer automatisierten Zertifizierung und Rezertifizierung von Software anhand des Beispiels des Industrial Data Space erörtert.

Zunächst wird in dieser Arbeit der Industrial Data Space vorgestellt und dessen Eigenschaften charakterisiert und anschließend die Anforderungen an eine Zertifizierung erörtert. Auf diese Erörterung folgend wird gezeigt, welche dieser Anforderungen bereits von bestehenden Technologien von CARiSMA abgedeckt werden und nachfolgend ein Konzept für eine Erstzertifizierung sowie später eine Erweiterung um eine Rezertifizierung vorgestellt. Im letzten Abschnitt werden die Vorteile, Herausforderungen sowie unser Ansatz für eine automatisierte (Re-)Zertifizierung zusammengefasst.

Der Industrial Data Space

Der Industrial Data Space ermöglicht den sicheren und kontrollierbaren Austausch von Daten zwischen Unternehmen mit dem Ziel der Analyse von Daten aus heterogenen Quellen, um auf dieser Basis neue Services und Produkte entwickeln zu können. Entsprechend Abbildung 1 wird an einer zentralen Stelle die Verfügbarkeit sämtliches im Industrial Data Space vorhandenen Wissen, aus dem digitalem Vertrieb, der digitaler Produktion, von Konsumenten sowie freiverfügbares Wissen, zusammengeführt und kann über diese Instanz abgefragt werden.

Abhängig von der Art des Wissens sowie den Präferenzen des Inhabers, steht das Wissen allen Teilnehmern am Industrial Data Space entweder entgeltlich oder unentgeltlich zur Verfügung. Die Verwendung

und Bereitstellung der Daten kann dabei an weitere Auflagen, wie z.B. Sicherheitsstandards bei der Datenverarbeitung, geknüpft sein oder Restriktionen unterliegen, wie z.B. das Verbot der Zusammenführung bestimmter Daten.

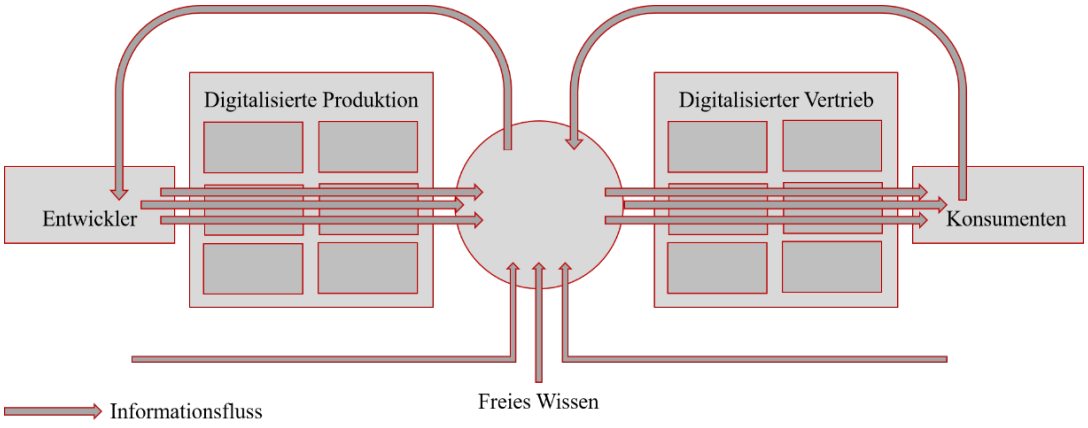


Abbildung 1: Informationsfluss im Industrial Data Space

Auch wenn Daten im Industrial Data Space vornehmlich zwischen Servern und PCs der einzelnen beteiligten Firmen, unter Zuhilfenahme einer zentralen Instanz, ausgetauscht werden, können diese lokalen Ressourcen bei Bedarf zusätzlich um in der Cloud angesiedelte Ressourcen erweitert werden.

Die im Rahmen eines Cloud-Computing-Angebots angemieteten Ressourcen, unterteilen sich in drei Ebenen. Zunächst bietet Infrastructure-as-a-Service (IaaS) das Anmieten von Hardware und einer minimalen Infrastruktur. Platform-as-a-Service (PaaS) stellt ein Betriebssystem zur Verfügung, auf dem Software eingespielt werden kann, während Software-as-a-Service (SaaS) nur die Verarbeitung oder Lagerung von Daten anbietet. Die Risiken hängen stark von der verwendeten Ebene, sowie dem verwendeten Verteilungs-Modell (z.B. öffentlich, privat, intern, extern) ab. Im Industrial Data Space können neben selbst betriebenen Rechenzentren alle drei Ebenen zum Einsatz kommen und müssen in entsprechenden Arbeiten berücksichtigt werden.

Die am Industrial Data Space beteiligten Unternehmen und Forschungseinrichtungen konzentrieren ihre Arbeit hauptsächlich auf die Erstellung von Anforderungskatalogen, Richtlinien und Verträgen, wobei die Umsetzung dieser den jeweiligen Zertifizierungsstellen und Dienstanbietern überlassen wird. Trotz der hohen Komplexität und der Notwendigkeit zur Berücksichtigung unterschiedlichster Anforderungen sowie Anhängigkeiten wird mangels ausreichender Werkzeugunterstützung vor allem in der Zertifizierung auf eine rein manuelle Kontrolle der zu zertifizierenden Artefakte gesetzt.

Weiterhin liegen für den Industrial Data Space als neue Idee bisher nur wenige Forschungsergebnisse vor. Es lassen sich jedoch einige der auf den Ansatz des Cloud-Computing bezogenen Forschungsergebnisse auf den Industrial Data Space übertragen. Zu berücksichtigen ist dabei, dass es sich beim Cloud-Computing im Vergleich zum Industrial Data Space eher um einen offenen Ansatz handelt, der teils mit geringeren, vor allem aber anderen, Sicherheitsanforderungen verbundenen ist.

Vom Cloud-Computing auf den Industrial Data Space lassen sich vor allem Arbeiten zu Kommunikationssicherheit und zur Identifizierung der an der Kommunikation beteiligten Parteien übertragen. Ein zusätzlicher Fokus im Industrial Data Space liegt auf Nutzungsbestimmungen der zwischen Unternehmen ausgetauschten Daten.

Sicherheitszertifizierung im Industrial Data Space

Bei unternehmensübergreifenden Geschäftsprozessen im Industrial Data Space und dem damit verbundenen Austausch von Daten sind verschiedene Compliance-Auflagen zu berücksichtigen. Es muss geprüft werden, welche Daten und in welcher Form das Unternehmen verlassen dürfen sowie zu welchen Zwecken diese Daten verwendet werden dürfen. Dürfen die jeweiligen Daten das Unternehmen verlassen, so kann z.B. abhängig vom Verwendungszweck eine Anonymisierung notwendig sein. Sämtliche in diesem Prozess beteiligten Komponenten von der Entscheidungsfindung, über die Anonymisierung und Übertragung bis zur Datenverarbeitung müssen per Zertifikat belegen, dass diese die jeweils gültigen Richtlinien und Normen einhalten [2, 3, 4].

Allgemein werden solche Zertifizierungen in der Industrie nach manueller Analyse unter Zuhilfenahme von normalsprachlichen Leitfäden oder Prüfkatalogen erstellt. Speziell zur Zertifizierung von Software-as-a-Service-Lösungen entwickelt der Verband der Cloud-Services-Industrie in Deutschland (EuroCloud Deutschland eco) ein Gütesiegel und einen zugehöri-

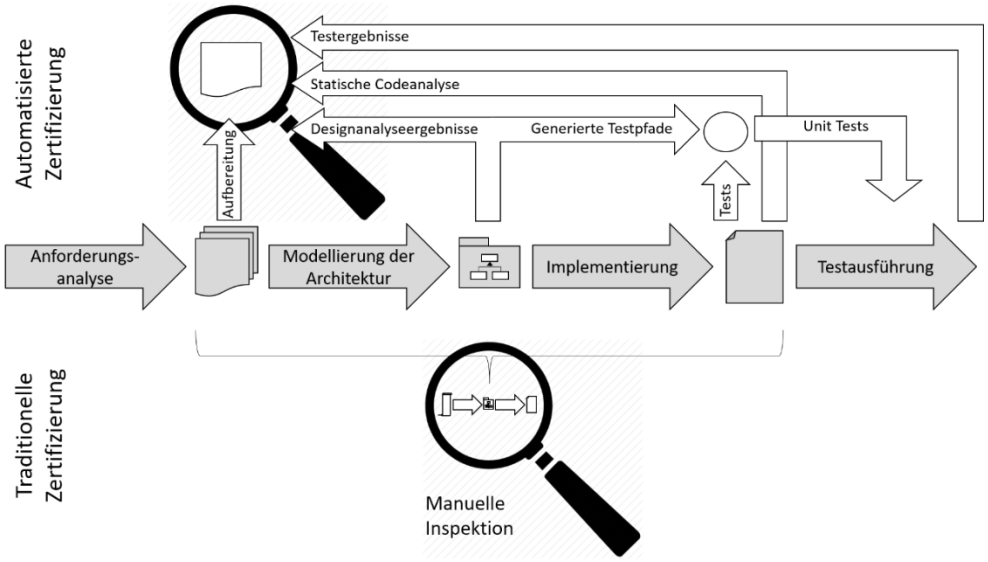


Abbildung 2: Manueller und Automatisierter Zertifizierungsprozess

gen Prüfkatalog. Vergleichbare Zertifizierungen werden vom Industrial Data Space e.V. für einen Datenaustausch zwischen Unternehmen angestrebt und zurzeit entwickelt [5].

Relevante allgemeinere Sicherheitsstandards für Cloud-Computing, den Industrial Data Space und andere IT-Anwendungen sind das „Statement on Auditing Standards (SAS) Nummer 70 Typ II“ und das ISO-Zertifikat 27001. Medizinische Produkte müssen z.B. neben diesen Standards auch nach der EU Richtlinie 93/42/EWG bzw. deren nationalen Umsetzungen zertifiziert werden.

Aus diesen Vorbildern und gesetzlichen Standards ergibt sich die Notwendigkeit einer Sicherheitszertifizierung der verwendeten Komponenten als wichtiger Bestandteil des Industrial Data Space, um eine sichere Kommunikation sowie Datenaustausch zu ermöglichen.

Für solch eine Sicherheitszertifizierung im Industrial Data Space wurden von der Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V. sechs Hauptaspekte identifiziert [5]:

Verbindungssicherheit gegen Manipulation und Abhören der übertragenen Daten.

Identitätsnachweis zwischen den jeweiligen Kommunikationspartnern.

Datennutzungskontrolle zur Einhaltung von Standards zur sicheren Speicherung und Verarbeitung sowie Richtlinien zur Nutzungsdauer und Weitergabe von bereitgestellten Daten.

Sichere Ausführungsumgebung zur Einhaltung von Sicherheitsleveln auf unterschiedlichen Plattformen.

Remote Attestation der Einhaltung von Löschungsfristen und der Feststellung eines vertrauenswürdigen Zustandes des Gegenübers.

Applikation Layer Virtualisierung für die Auslagerung von Teilsystemen in die Cloud.

Zur Zertifizierung einer Komponente unter Berücksichtigung dieser Aspekte müssen vielfältige Eigenschaften berücksichtigt werden. Wird solch eine Zertifizierung ohne eine Werkzeugunterstützung durchgeführt, ergibt sich daraus ein langwieriger und fehleranfälligerer Prozess.

(Re-)Zertifizierung mittels CARiSMA

Die von uns entwickelten Techniken zur Modellierung und Verifizierung von Sicherheitsanforderungen auf UML-Modellen und konkreten Implementierungen der jeweiligen Modelle sind geeignet, um ein umfassendes Konzept zur Absicherung von digitalen Geräten und Netzwerken zu erstellen.

Ein konkreter Ansatz zur Modellierung komplexer Internet-basierter Systeme, der in unserem Sicherheitsmodellierungswerkzeug CARiSMA verwendet wird, ist UMLsec [6]. Im UMLsec-Ansatz können Software-Design-Modelle, die mit der Unified Modeling Language (UML) erstellt werden, wie in Abbildung 3 mit sicherheitsrelevanten Informationen annotiert werden.

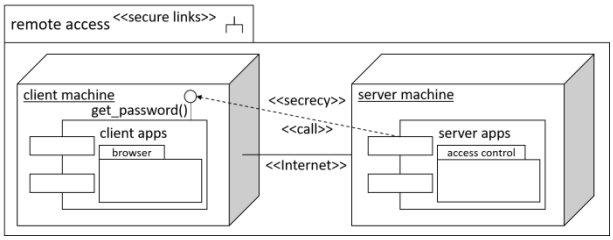


Abbildung 3: UMLsec Beispiel Secure Links

Mittels des UMLsec-Stereotyps <<secure links>> wird in dem Beispiel aus Abbildung 3 die Anforderung spezifiziert, dass kritische Kommunikation, wie z.B. die Passwortabfrage zwischen Client und Server im Industrial Data Space, über abgesicherte Verbindungen erfolgen muss. Die kritische Kommunikation

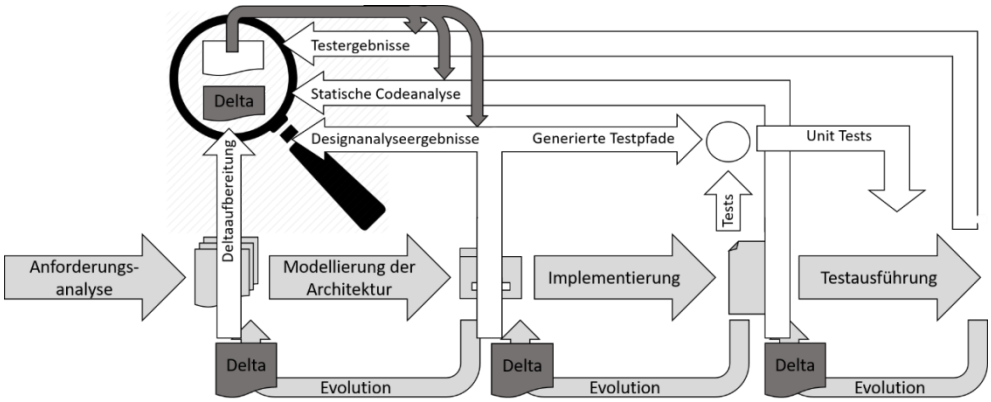


Abbildung 4: Rezertifizierung mittels CARiSMA

wird mittels des Stereotyps <<secrecy>> gekennzeichnet.

Diese mittels UMLsec um Sicherheitsanforderungen erweiterten UML-Modelle können nachfolgend mit automatischen Werkzeugen wie CARiSMA auf Einhaltung der mittels UMLsec spezifizierten Sicherheitsanforderungen überprüft [7] und ggf. vorhandene Schwachstellen automatisch korrigiert werden [8].

In dem Beispiel aus Abbildung 3 ist eine <<Internet>> Verbindung nicht ausreichen, um die <<secrecy>> Anforderung des Passwortabrufes, durch einen Server aus der digitalen Produktion des Industrial Data Spaces an einen Client aus dem digitalen Vertrieb, zu erfüllen.

Weiterhin kann durch die Generierung von Sicherheits-Monitoren aus den in UMLsec spezifizierten Sicherheitsanforderungen die Sicherheit auch zur Laufzeit überwacht werden [9]. Ebenfalls ist eine Generierung von Testpfaden zur Überprüfung der Sicherheitsanforderungen mittels des selben Mechanismus möglich.

Erstzertifizierung: Während bei einer traditionellen manuellen Zertifizierung entsprechend Abbildung 2 alle Entwicklungsschritte ganzheitlich von einem Menschen betrachtet werden, werden in CARiSMA die Ergebnisse einzelner Entwicklungsschritte automatisiert ausgewertet und aufbereitet und z.B. natürlich sprachliche Fragen zu Privatsphärenpräferenzen der Anwender der jeweiligen Plattform generiert.

Zukünftig können in diesen aufbereiteten Daten sowohl die existierenden Ergebnisse der Anforderungsanalyse, wie z.B. für diese Zertifizierung relevante Standards und Sicherheitsanforderungen, die Berücksichtigung und Einhaltung der Sicherheitsanforderungen in Softwaredesign bzw. Softwarearchitektur und in dessen Implementierung sowie letztendlich die Auswertung von Nutzertests und aus den Sicherheitsspezifikationen generierter Tests zusammengeführt werden.

Dabei bilden die mittels UMLsec modellierbaren Sicherheitsanforderungen einen großen Teil der für die Sicherheitszertifizierung im Industrial Data Space benötigten Kriterien ab. Durch eine Anwendung der mit UMLsec verbundenen Werkzeuge auf die Herausforderung der Sicherheitszertifizierung im Industrial Data Space kann auch die geforderte Automatisierung des Zertifizierungsprozesses erreicht werden.

Mittels der Spezifikation und Analyse von Sicherheitsanforderungen in UMLsec wird sowohl die Architekturebene als auch durch die Generierung von Sicherheitsmonitoren und Testpfaden große Teile der Implementierungsebne abgedeckt.

Mittels der generierten Testpfade können zukünftig weiterhin die vorhandenen Nutzertests auf Abdeckung der Testpfade geprüft werden sowie gegebenenfalls zusätzliche Tests generiert werden. Die in dieser Erweiterung generierte Testfälle erweitern die vorhandenen Nutzertests um sicherheitsspezifische Testfälle und werden gemeinsam mit den Nutzertest ausgeführt und die Testergebnisse für die Zertifizierung aufbereitet.

Die Aufbereitung der Testausführung kann dabei mehr als eine Statistik über erfolgreich ausgeführte Testfälle enthalten. Durch die explizit unter dem Gesichtspunkt der Sicherheit generierten Testfälle, können in der Aufbereitung der Testergebnisse explizite Aussagen über die Eigenschaften der Implementierung bezüglich bestimmter Sicherheitsanforderungen getroffen werden.

Für die Identifizierung relevanter Regulierungen bei der Auswertung der Anforderungsanalyse werden diese als Ontologie dargestellt. Diese Ontologie kann genutzt werden um ein gegebenes System bezüglich der Einhaltung von Compliance-Anforderungen zu untersuchen. Zusätzlich können mittels einer erweiterten Werkzeugunterstützung Systemmodelle auf bestimmte, als relevant identifizierte, Eigenschaften geprüft werden [10].

Rezertifizierung: Die separate Aufbereitung einzelner Entwicklungsschritte im vorgeschlagenen Zertifizierungsprozess ermöglicht die in Abbildung 4 in dunkelgrau dargestellte Erweiterung des Zertifizierungsansatzes um eine Rezertifizierung eines zuvor mittels dieses Zertifizierungsprozesses zertifizierten

Programms, in der von einem menschlichen Gutachter nur noch die aufbereiteten Änderungen begutachtet werden müssen.

Für jeden einzelnen Schritt des Entwicklungsablaufes wird in diesem Rezertifizierungsansatz ein Delta generiert, welches die vorgenommenen Änderungen beinhaltet. Anhand dieses Deltas wird der jeweils vorangehende Schritt neu bewertet und ggf. z.B. neue einzuhaltende Normen, Sicherheitsanforderungen, Testpfade oder Sicherheitsmonitore generiert und in den nachfolgenden Schritt propagiert.

Bei der Präsentation der Ergebnisse können diese Deltas und die Ergebnisse der erweiterten Analysen in einer Form separat aufbereitet werden, die es ermöglich nicht erneut alle Bestandteile des Programms und Entwicklungsprozesses für die Rezertifizierung manuell betrachten zu müssen.

Dazu werden die Ergebnisse vorheriger Zertifizierungen mit den Ergebnissen der Rezertifizierung kombiniert und wiederum ein Delta berechnet, welches für eine manuelle Begutachtung präsentiert wird.

Zusammenfassung

Durch eine starke Automatisierung des Zertifizierungs- und vor allem des Rezertifizierungsprozesses kann nicht nur die Effizienz der Zertifizierung gesteigert werden und eine Zertifizierung dadurch für deutlich mehr Produkte verfügbar gemacht werden. Durch eine zielgerichtete Aufbereitung der Daten sowie eine verbesserte Berücksichtigung aller Abhängigkeiten kann auch die Effektivität der Zertifizierung gesteigert werden.

Auf großen Plattformen, wie dem Industrial Data Space ist es für einen Menschen nahezu unmöglich alle Abhängigkeiten sowie deren Auswirkungen zu erkennen und zu beurteilen. Eine Werkzeugunterstützung ist an dieser Stelle zwingend notwendig. Der von uns vorgestellte Ansatz erfüllt die Anforderungen zur Berücksichtigung von Abhängigkeiten und Auswirkungen, die an solch eine Werkzeugunterstützung gestellt werden.

Die Strukturierung des vorgestellten Zertifizierungsablaufes ermöglicht zum einen die Berücksichtigung von Abhängigkeiten innerhalb der zu zertifizierenden Plattform aber vor allem auch zwischen den einzelnen Schritten des Entwicklungsprozesses.

Die bestehenden Modellierungs- und Analysefähigkeiten von CARiSMA decken bereits große Teile der benötigten Eigenschaften und Fähigkeiten ab und können mit kleinen Erweiterungen den vorgestellten (Re-)Zertifizierungsansatz unterstützen.

Dabei ist von Vorteil, dass von CARiSMA in der Softwareentwicklung weit verbreitete und standardisierte Artefakte, wie z.B. UML-Modelle, verwendet werden. Dies ermöglicht es CARiSMA ohne große Anpassungen in bestehende Entwicklungsabläufe zu integrieren.

Zusammenfassend besteht in der Industrie ein großer Bedarf an der Automatisierung von Sicherheitszertifizierungen. Unserer Sicherheitsmodellierungs- und Analysewerkzeug CARiSMA deckt bereits große Bestandteile solch einer automatisierten Zertifizierung ab und kann mit geringem Aufwand entsprechend des in dieser Arbeit vorgestellten Konzepts eines (Re-)Zertifizierungs-prozesses erweitert werden.

Literaturverzeichnis

[1] Industrial Data Space e. V., [Online]. Available: http://www.industrialdataspace.org.

[2] S. Ahmadian, F. Coerschulte, J. Jürjens: "Supporting the Security Certification of Cloud-Computing-Infrastructures (invited paper)" in *International Symposium on Business Modeling and Software Design*, 2015.

[3] J. Jürjens: "Geschäftsprozesse in der Cloud - aber sicher ! (… und compliant)," in *Software Engineering + Architectures*, 2014.

[4] S. Wenzel, C. Wessel, T. Humberg, J. Jürjens: "Securing Processes for Outsourcing into the Cloud," in *International Conference on Cloud Computing and Services Science*, 2012.

[5] B. Otto, J. Jürjens, J. Schon, S. Auer, N. Menz, S. Wenzel, J. Cirullies: "Industrial Data Space - Digitale Souveränität über Daten," Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V., München, 2016.

[6] J. Jürjens: “Secure Systems Development with UML”, Heidelberg: Springer, 2005.

[7] J. Jürjens: "Sound Methods and Effective Tools for Model-based Security Engineering with UML," in *International Conference on Software Engineering*, 2005.

[8] J. Jürjens: "Automated Security Hardening for Evolving UML Models," in *International Conference on Software Engineering*, 2011.

[9] A. Bauer, J. Jürjens, Y. Yu: "Run-Time Security Traceability for Evolving Systems," *The Computer Journal*, vol. 54, no. 1, pp. 58--87, 2011.

[10] T. Humberg, C. Wessel, D. Poggenpohl, S. Wenzel, T. Ruroth, J. Jürjens: "Using Ontologies to Analyze Compliance Requirements of Cloud-Based Processes," in *Cloud Computing and Services Science (selected best papers)*, 2014.

[11] D. C. Petriu, C. M. Woodside, D. B. Petriu, J. Xu, T. Israr, G. Georg, R.B. France, J.M. Bieman, S.H. Houmb, J. Jürjens: "Performance analysis of security aspects in UML models". WOSP 2007: 91-102

[12] J. Jürjens: „Modelling Audit Security for Smart-Cart Payment Schemes with UML-SEC". SEC 2001: 93-108

[13] F. Dupressoir, A.D. Gordon, J. Jürjens, D.A. Naumann: "Guiding a general-purpose C verifier to prove cryptographic protocols". Journal of Computer Security 22(5): 823-866 (2014)

[14] S.H. Houmb, G. Georg, J. Jürjens, R. France: "An integrated security verification and security solution design trade-off analysis approach". In: Integrating Security and Software Engineering: Advances and Future Visions, 2007. Pages 190-219.

[15] K. Schneider, E. Knauss, S.H. Houmb, S. Islam, J. Jürjens: „Enhancing security requirements engineering by organizational learning". Requir. Eng. 17(1): 35-56 (2012)

[16] S. Islam, H. Mouratidis, J. Jürjens: "A framework to support alignment of secure software engineering with legal regulations". Software and System Modeling 10(3): 369-394 (2011)

[17] J. Jürjens: "Model-Based Security Engineering with UML". FOSAD 2004: 42-77

[18] S. Höhn, J. Jürjens: „Rubacon: automated support for model-based compliance engineering". ICSE 2008: 875-878

[19] A. Bauer, J. Jürjens: „Security protocols, properties, and their monitoring". SESS 2008: 33-40

[20] J. Jürjens, J. Schreck, Y. Yu: „Automated Analysis of Permission-Based Security Using UMLsec". FASE 2008: 292-295

Autoren

Sven Peldszus

Sven Peldszus ist wissenschaftlicher Mitarbeiter in der Arbeitsgruppe Software-Engineering von Prof. Dr. Jan Jürjens an der Universität Koblenz-Landau. Sein Fokus liegt auf der statischen Softwareanalyse.

speldszus@uni-koblenz.de

Prof. Dr. Jan Jürjens

Jan Jürjens ist Professor für Software-Engineering an der Universität Koblenz-Landau sowie Director Research Projects am Fraunhofer ISST in Dortmund. Er ist Autor des Buches "Secure Systems Development with UML" (Springer-Verlag 2005, chinesische Übersetzung 2009).

http://jan.jurjens.de

Competition-Driven Quality Boost

How to drive software quality to the next level

High software quality is inherently important in order to deliver products that meet the standards of involved customers and stakeholders in general. Static and dynamic metrics and measurements allow for monitoring software and to indicate the quality at any level of the software development process. These methods for monitoring software – and going even further, for example by introducing quality gates on build systems – are well evolved and nowadays usual in professional software development. Within this paper, we introduce competition in addition as motivational factor to drive software quality to the next level. We employ standard functionality and well evolved metrics, incorporated with company-wide competition in order to significantly improve software quality in our products.

Introduction and Motivation

In general, the term "software quality" is very widespread. Different quality models have been introduced in the recent decades, considering not only the quality of the developed product, but also the quality of the involved development processes. One commonly accepted model is the international standard ISO/IEC 9126 (that has been already replaced by ISO/IEC 25010:2011). There, software quality is classified in a structured set of characteristics (and sub-characteristics), including for example:

Functionality (including Suitability, Accuracy, Interoperability, etc.)

Reliability (including Fault Tolerance, Recoverability, etc.)

Usability (including Understandability, Learnability, etc.)

Efficiency (including Resource Utilization, Time Behavior, etc.)

Maintainability (including Changeability, Testability, Stability, etc.)

Portability (including Adaptability, Replace-ability, etc.)

Static and dynamic metrics that measure software quality in terms of the above listed characteristics include standards as for example provided by the SonarQube Platform, like:

Complexity (which is usually the cyclomatic complexity, also known as McCabe metric). The complexity is usually measured as relative factor considering (i) the complexity per class, (ii) the complexity per file or - finer grained – (iii) the complexity per method.

Code Duplication: following the book "Clean Code" written by Robert C. Martin, "*code duplication may be the root of all evil in software*", duplication in software has to be desperately avoided. Basically, code duplication can also be measured on different levels, e.g.: (i) duplicated blocks of source lines (whereas the definition how many duplicated lines have to be considered as duplication has to be met), (ii) duplicated files, (iii) duplicated methods, or (iv) the density of duplicated lines considering the amount of duplicated lines relative to the total amount of source lines.

Issues describe the number of violations that are registered in total by SonarQube. Those can be further categorized in (i) new issues, (ii) issues in total, (iii) false positive issues, (iv) open issues, (v) weighted issues, etc.

Size: considers different metrics to capture the general characteristics of a software solution. This includes - besides others – (i) the number of classes, (ii) the number of files, (iii) the lines of code in total, (iv) number of methods, etc. These metrics have generally an informing character.

Technical Debts are calculated by SonarQube and define how long it would take to correct all open issues. By defining a debt per issue (in time needed to fix) the sum can be calculated and can be taken as good indication of the current software quality state.

Tests: considers the parts of the software product that are covered by tests. Specific metrics include (i) line coverage, (ii) condition coverage (considers boolean expressions), (iii) total number of unit tests, (iv) uncovered lines (total number of lines of code that are not covered by unit tests), etc.

These metrics and methods for quantifying and handling of software quality are all well-known and widespread. Experienced software developers can handle such metrics instantly and optimize them towards high software quality.

Nevertheless, since we at ecx.io[1] have very well educated professional developers that know how to achieve high software quality, we thought about methods how to push the understanding of quality – and thus the quality of the delivered products themselves – to the next level. Furthermore, we strongly have the intent to make quality quantifiable and easy understandable. In order to do so, we believe in the hypothesis that by incorporating a competition in achieving high software quality, our products and the software development process can be significantly improved and moved up to the next level. The hypothesis can be summarized to:

> Hypothesis: *Competition-driven software development boosts software quality to the next level.*

Therefore, we introduced the Quality Improvement Ranking, that provides a real-time ranking of quality improvements of the specific projects. We honor the development team that improves most during the period of a month compared to other projects. Following the idea that competition can release motivation and unlock high potentials, we can observe a dramatic gain of momentum and a significant increase of motivation to win the Quality Improvement Ranking.

> "*Competition helps us to get and stay motivated. It helps us generate new energy when we are stagnated.*" – Matt McWilliams

By providing a web-based ranking dashboard that is automatically updated upon each build process (to clarify, the metrics that are received from Sonar are updated upon each build, the one metric that is generated out of Jira is received once per day), we encourage our developers to outgrow and excel themselves when developing software. Some teams even developed such high motivation to win the quality ranking that they even battled eagerly against other development teams.

The remainder of this paper explains our approach and hypothesis in detail, how we handle software quality in general at ecx.io, which metrics and technologies are used for quantifying relevant metrics for the software quality improvement ranking, and – most importantly – which results we were able to achieve with competition-driven development.

Software Quality Handling at ecx.io

We at ecx.io have all common standards and technologies available to ensure high software quality. Beginning from state of the art infrastructure on the local development machines, to state of the art build server technologies (e.g. Jenkins) to GIT as versioning system. Furthermore, to capture numeric measurements in form of software quality metrics, we heavily use the Sonar platform (with customized and optimized quality profiles per programming language) in our Continuous Integration process that evaluates software projects upon each build. To sum up, unit tests are trigged automatically upon each build process.

These are common technologies that are very widespread in professional software development. Unfortunately, very often, the amount of information describing the state of the software quality that is provided as feedback to the developer is simply too much to utilize in daily business. According to the KISS principle (Keep It Simple, Stupid), we narrowed the quality metrics down to four significant KPIs (Key Performance Indicators). Nevertheless, the full spectrum of quality information per project is available for each employee via the SonarQube Platform. Figure 1 shows an example of a Sonar dashboard for a specific project.

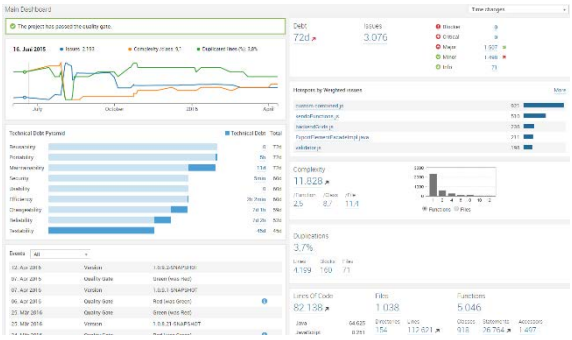


Figure 2: Exemplary Screenshot of a Sonar Dashboard as used at ecx.io.

The four metrics selected to be considered within the software quality improvement ranking (weighted open issues, complexity, duplication and sonar issues) capture very well the current state of a solution. On the one hand we provide easy to understand static quality metrics in form of the complexity and the duplication, and on the other hand an overview about the current quality state from a higher abstracted level by providing the weighted open issues (as tracked for example in JIRA) and the sonar issues. Figure 2 provides an overview about the presentation of the 4 metrics for an exemplary solution. The yellow and red

[1] See https://www.ecx.io/

area in the top left image illustrate the quality indicator, whereas green is in the range from 0-50, yellow in the range 51-100 and red if the total number of weighted open issues is greater than 101. Similar quality indicators are also introduced for the other metrics.

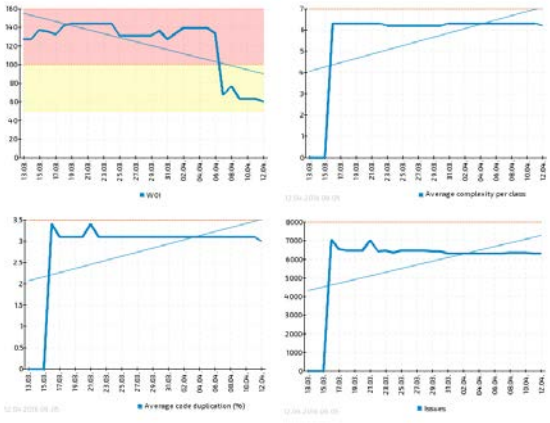


Figure 3: Exemplary illustration of the 4 relevant metrics.

The Quality Improvement Ranking as illustrated in Figure 3 not only provides a current state regarding the 4 metrics, but also combines them to an average number that is used to rank the projects in terms of improvements. By hovering over a value the corresponding image as shown in Figure 2 is rendered and provided. The number that is displayed in the Quality Improvement Ranking – for example for the weighted open issues – is a relative number over time. For example a complexity value of -10% indicates that the complexity of the solution compared to the previous period of a month has been improved by 10%. Every month (i.e., the beginning of a new period), all values are resetted to zero. So every team has a new chance to win the quality improvement ranking. Green indicates an improvement, red a degradation of the specific value. The current rank of the project as well as the rank of the project before the latest update of the dashboard itself is shown at the very left.

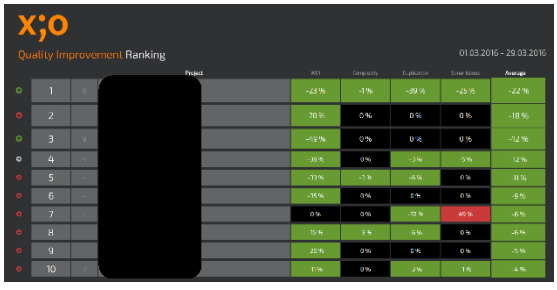


Figure 4: The Quality Improvement Ranking Dashboard (the names of the customers/projects are blacked out).

As it can be seen in Figure 3, the Quality Improvement Ranking is composed of the four metrics (i.e., Key Performance Indicators - KPI) as summarized in the following Table 1.

Quality KPI	Short Description
WOI	Weighted Open Issues
Complexity	The average cyclomatic complexity per class
Duplication	The average code duplication
Sonar Issues	The count of issues registered by Sonar

Table 1: Quality KPI Overview

The numerous value for each considered KPI is used for calculating an average value for the ranking. In detail, the used KPIs are the following:

WOI: is a trend analysis of weighted unresolved internal and external bugs over time. This value can be extracted directly from the corresponding JIRA project. The weighting respects the characteristic of the issue. Within our JIRA projects we have defined weighting factors for different types of issues: (i) a top priority issue is weighted with factor 10, (ii) a "normal" priority issue is weighted with factor 5, (iii) a standard issues is weighted with factor 3 and (iv) a minor issues is weighted with factor 1. If for example 18 issues are open, whereas 2 are top-priority issues, 5 are priority issues, 8 are standard issues and 3 are minor issues, the WOI is 72. If the development team manages to reduce this WOI value to 54, this would be an improvement of 25%.

Complexity: The average cyclomatic complexity per class as calculated by Sonar. Basically, this metric is a quantitative measure of the number of linearly independent paths through a source code. Simply spoken: the more nested if- and switch statements, the worse (i.e. the higher the cyclomatic complexity). Relevant for the quality improvement ranking is the average complexity per class. A value less than 10 is acceptable, a value between 10 and 14 is marked as yellow, and a value higher than 15 is marked as red in the corresponding figure as shown in Figure 3. This classification is used to provide a quick overview about the state of this very metric. If for example the team manages to change the complexity value from 10 to 9.5, this would mean an improvement of 5% within the quality improvement ranking.

Duplication: The average number of duplicated lines of code in the project. Considering duplication is inherently important for software quality, since "duplication may be the root of all evil in software" – as stated by Robert C. Martin in his book "Clean Code – A Handbook of Agile Software Craftsmanship". The amount of lines that are considered as being duplicated are defined in the Sonar Profiles per Language. The average code duplication as calculated by Sonar defines the relative number of lines of code that

are duplicated with respect to the total lines of code. A value between 0 and 5% is acceptable (green), between 5% and 8% the duplication metric is marked as yellow, a value higher than 8% is marked as red in terms of the quality indicator.

Sonar Issues: The total number of issues appearing in Sonar as defined within the quality profiles. Sonar issues are categorized in (i) blocker, (ii) critical, (iii) major, (iv) minor and (v) info.

The calculated values for each four KPIs are summarized and divided by four in order to gather the average value that is used to rank the improvements of the projects. So by executing targeted actions upon the software project by the develop team members, it is possible to push a project up in the quality improvement ranking. Generally, this calculation is done instantly upon triggered build processes within the continuous integration process – resulting in daily (or even hourly) updates of the Quality Improvement Ranking.

The next section summarizes the results of the competition regarding the quality improvements, and also discusses interesting aspects regarding the observed motivation, the gained momentum and – most importantly – the ambition that teams were able to develop in order to push the own project up in the quality improvement ranking resulting in an improved quality of our delivered products.

Quality Improvement Results

With the first skepticism of the developer team members being dissipated, we currently can observe a remarkable drive and motivation gain trying to win the quality improvement ranking. The team that is ranked first at the end of the month wins an award. Figure 4 shows the proud winners of the month November 2015 receiving the quality improvement award.

Figure 5: Quality Improvement winners receiving the award

Furthermore, by introducing a defects first policy, not only the overall software quality has been improved but also the customer acceptance by fast responses to occurring errors.

The effect that we expected is obvious: the general quality of our software projects increases remarkably. The raw numbers to underpin the success of our approach show very good results in the software quality of our projects. The following Table 2 shows the average evolution of the three quality KPIs Duplication, Complexity and Sonar Issues over time since introduction of our approach of the quality improvement ranking. As it can be seen, the average values of representative projects (which are in total 37 projects) are decreasing. The averaged value for all 37 projects for the Duplication metric was improved from 8,16% to 7,41%; the value for the cyclomatic complexity from 10,34% to 10,25%; the value for sonar issues from the average value of 1499 to 1193. The fourth quality KPI – the WOI cannot be considered here, since the JIRA issues from the past cannot be easily surveyed.

Quality KPI	Before (Average)	After (Average)
Duplication	8,16%	7,41%
Complexity	10,34	10,25
Sonar Issues	1499	1193

Table 2: Average Evolution of Quality KPIs

Selected projects show tremendous improvements in the different metrics. The most improvements regarding the three metrics are the following (achieved by different projects):

Duplication: from 8,5% to 1,6%

Complexity: from 17,5 to 14,7

Sonar Issues: from 5775 to 3733

The hypothesis can be stated of being right. By introducing the quality improvement ranking and the award, we are able to significantly improve the quality of our delivered projects, whereas this process is still ongoing. The following section closes the paper with a summary and conclusion.

Summary and Conclusion

To summarize our approach, we utilize standard software quality analysis and metrics together with a fully

integrated continuous integration pipeline. The up-to-date information about the current state of the software quality of a product is available for the members of the development team at any time. Nevertheless, we observed that simply providing a SonarQube Platform together with the commonly known dashboards and the usual quality gates to avoid degenerations of the software is sufficient for a standard level of software quality.

In order to push the software quality level within our company ecx.io to the next level, we follow the approach of competition-driven development. We utilize four easy-to-understand metrics that are gathered from Sonar and JIRA in order to capture the current quality state of a certain product. These four metrics are combined to one numeric value indicating if the quality of the affected projects improves, stagnates or degrades. To motivate the development teams, a monthly winner of a quality improvement competition is elected. This competition boosts the quality in our projects to the next level and shows tremendous improvements in software quality proving our hypothesis to be absolutely correct.

Bibliography

McCabe, A Complexity Measure, IEEE Transactions on Software Engineering: 308–320, Dec. 1976

Martin, Clean Code – A Handbook of Agile Software Craftsmanship, Prentice Hall, Pearson Education, Inc., 2009

ISO/IEC 9126:2001, Software Engineering, Product Quality, Part 1: Quality model, 2001

ISO/IEC 25010:2011, Systems and Software Engineering, Systems and software quality requirements and evaluation, 2011

McWilliams, Competition as Motivation – What is Pushing you to succeed? (Blog Post), http://www.mattmcwilliams.com/competition-as-motivation-what-is-pushing-you-to-succeed/

SonarQube Platform,
http://docs.sonarqube.org/display/HOME/SonarQube+Platform

Authors

DI Dr. Marc Kurz,
Senior Developer working for ecx.io austria (an IBM company).
Marc is a Senior Developer and Technical Architect in the .NET area and working for exc.io since December 2014. He studied computer science and holds a master and doctorate degree from the University of Linz. Marc is a software passionate and expert in software quality, software engineering and generally software architectures.
marc.kurz@ecx.io

David Stöger, MSc,
Director Platform Technology working for ecx.io austria (an IBM company).
David is leading the development teams at ecx.io austria and also supervising the Croatian development teams. He started at ecx.io back in 2008, beginning in the Java and Adobe Flex area but quickly moved over to .NET web application development. David has a Master's Degree in Software Engineering from the University of Applied Sciences in Hagenberg. David is a technical professional with strong skills in team management and team collaboration.
david.stoeger@ecx.io

1, 2, 3 – Build!

Continuous Integration for Mobile Applications

Continuous Integration is a well-established process for supporting the development, test and release of applications. However, when developing mobile applications, many companies still rely on manual processes that are expensive and error-prone, because a few years ago, the tools were not ready or required specialists for setting them up.

This article presents tools, frameworks and services that have been made available recently to solve this issue. By making them easy to use, developers have no more excuse to not do Continuous Integration, even while facing some unique challenges of mobile app development such as testing on real devices.

Continuous Integration

Continuous Integration (CI) is the practice of merging all developer copies into a shared repository, possibly several times a day. The main idea is to prevent integration problems that originate from multiple developers working on separated copies of the same source-code for some time. Especially the merging can be simplified significantly if it happens on a small scale instead of merging largely diverged branches of the same source base. Typically a Continuous Integration server is in the loop and automatically checks the source-code that has been pushed into the repository by building it, exercising the tests and sending feedback to the developers within a short period of time. This short feedback-loop is critical, if something broke down, because the developer still knows the places that changed most recently, which limits the area, where bugs might have been introduced.

Using Continuous Integration has a wide range of benefits, namely (1) the early detection of integration bugs, (2) constant availability of a "current" build for testing and release, (3) reduced dependencies on individuals and (4) small losses if reverting to an earlier state is required.

Continuous Integration does not come for free. Especially setting up the infrastructure and getting the Continuous Integration server up and running can be tricky and takes some time. Such tools often cost money, but in my experience the costs for not using Continuous Integration always exceeded those costs by magnitudes.

That said, Continuous Integration is being used nowadays by almost all professional software developers, although the details on how to work with branches and how to merge individual changes to ensure that the master branch of the repository always stays green are still begin disputed.

Continuous Delivery

Taking the idea of Continuous Integration one step further, leads to the practice of Continuous Delivery, where not only the building and testing of the application is automated, but also the actual rollout of an application. This does not mean, that every change in the source-code repository necessarily leads to a new version of your app in the App store. But all the required steps like signing and uploading a binary package or updating screenshots are being performed automatically.

Tools like Visual Studio Team Services (VSTS) allow to define release definitions with several stages, e.g. automatic rollout to beta-users, staged rollout to 20% of all users and full rollout to all users. Each stage can have certain quality gates or require the approval from specific authorities.

As Continuous Delivery is based on Continuous Integration, it has the same advantages but adds the following benefits, namely (1) avoiding last-minute chaos before releasing a new version, (2) reproducible releases with version-controlled definitions and automated workflows, therefore (3) independence from individuals, (4) reduced costs for releasing and (5) dropped inhibition threshold for performing a release, as an actual release is not much more effort than a few clicks on a website.

So is possible, we should always strive for Continuous Delivery. But still, many companies do not have a Continuous Delivery process and heavily rely on manual steps being taken, whenever a new version of their software is released.

Continuous Delivery in practice

Most developers know the benefits of a Continuous Delivery process, but use some of the following excuses to justify that many steps of the release process are still done manually:

"This is just a small project. I can quickly test and upload it manually"

True, many projects start small – but eventually they grow bigger if they are successful. And even for really tiny projects with a single developer, the repetitive steps involved

in the release process are just boring and some steps might have been forgotten along the way.

"We have no time or budget to set up such services"

The time for setting up a cloud-service for building and testing a mobile application has been reduced dramatically by providers like Greenhouse that offer a streamlined process for setting up the server within minutes. And especially if setting those things up at the beginning of the project, a lot of time can be saved later on.

Cloud-services do cost some money, but most cloud-services offer free plans within certain limits. Only when exceeding those limits, additional packages can be purchased with prices that are still ridiculously low compared to the hourly rate of a developer.

"We don't have a server to run a CI on"

That is not a problem at all, because there are many commercial providers that provide the entire infrastructure for building mobile applications as a service. You don't have to do it on your own and I would not recommend doing so.

"Those services sound great, but we have strict NDAs and are not allowed to use cloud-services"

Strict NDAs do pose a problem, but very often exceptions can be negotiated with clients. And regarding the security concerns – remember that keeping your data safe and private is an essential core competence for any cloud-service provider. A break into their system could have devastating consequences for the company. If all that does not work out, you can still install on-premise versions of various CIs like Travis or VSTS.

"We only release twice a year, so the effort of setting up the Continuous Delivery process is not worth it"

Even if you release only twice a year, you still have the effort of setting up the process just once. And when you start automating this process, you might end up with shorter release cycles which gives you more flexibility and eventually happier customers because you can fix issues faster.

"I don't know how to set up Continuous Delivery for mobile application because there are some steps that cannot be automated."

Really? Just read on.

1 - Building your app automatically

Building mobile applications for the three mobile platforms Android, iOS and Windows unfortunately requires that you run more than one (virtual) machine, since iOS apps can only be built on Apple hardware and Windows UWP apps require Windows (which at least can be virtualized). If you can restrict yourself on the two major platforms Android and iOS, one Apple machine should suffice. Otherwise, a solution that uses multiple agents on different machines can be used. For example, Microsoft offers a platform-independent agent for VSTS.

Setting up an automated build with Greenhouse-CI is a matter of minutes. A straight-forward wizard guides the user through each step, automatically inspects the repository and executes all found tests, including Android Instrumentation Tests that require an emulator (see figure 1).

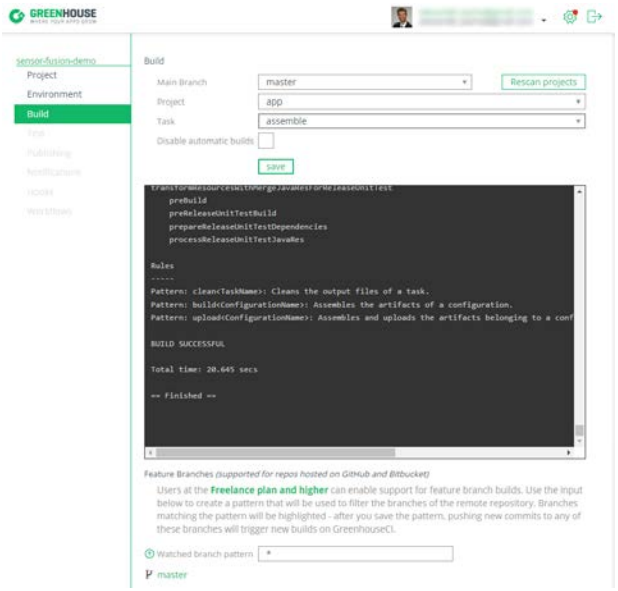


Figure 1: Greenhouse-CI wizard for setting up an automated build

Note that you should automatically set the version number of your application during the build to enable traceability. This can be achieved with a few lines inside a gradle file or with a separate build-step that runs a script.

Signing your app automatically

Signing an application is the process of applying a cryptographic signature to a binary package to enable users to verify that the source of an application is equal to its claimed source.

Signing Android applications is a simple process that can be included into the gradle script which assembles the app. All that is required is a keystore that can be generated with a single command from the shell containing a private key, a public key and some metadata. As there is no verification of the physical entity that created a keystore or the metadata specified within, everything that Google can guarantee, is that an updated version of an application has the same vendor as the originally installed application. Be aware of the fact that everybody could claim to be a member of a major company and –until reported to Google – could distribute malicious software over the Google Play store using that name.

Apple on the other side has very strict rules that have to be followed, before an application is granted permission to the App store. Basically Apple, and only Apple, can allow an application to run on an iOS device. Signing the application with a developer certificate (which gets issued by Apple) is only the first step of the entire procedure. After submitting an application, it is being reviewed and re-signed with Apple's security certificate. Only then it

can run on any iOS device. This leads to the necessity of special profiles – called provisioning profiles – that allow developers to run applications on their own devices. At least, Apple and Xamarin provide step-by-step tutorials on how to create the required certificates and provisioning profiles.

Another great tool for signing iOS applications is called *match*. It is accompanied by a guide of best practices on performing code signing when working as a team of developers.

2 - Testing your app automatically

That you write automated tests for your application should be a natural thing. Writing unit-tests for mobile applications is to no extend any different from writing unit-tests for desktop- or web-applications. The first difference occurs when you start creating more extensive integration-tests that instrument the lifecycle of your application and eventually call the native API. There are basically two approaches to handle this scenario: hide all such calls behind a mockable interface that is initialized accordingly in the tests to yield the desired results or use tools such as Robolectric that are able to handle native calls and process them without the need of an emulator.

The most challenging tests however are full-blown user-interface tests that mirror complete user-scenarios. The tools for writing such tests have seen tremendous improvements in the last few years. Tools like Espresso, Robotium, Xamarin.UITest or XCTest allow to write tests in that resemble the following pattern: *On a given view X, find the element that matches criteria Y and perform the action Z.* More concrete, this could be *onView(withText("OK")).perform(click()).* Note that the actual syntax might deviate slightly for each of the previously mentioned frameworks, but the general idea is the same for all of them.

Note that you can create those UI-tests even easier by using UI-test recorders that are available for Android, iOS and Xamarin.

Once a few tests have been created, a step can be added on the CI-server to automatically execute them, whenever a change is committed. Most CI-systems have predefined steps for unit-testing, firing up an emulator and executing the tests on a virtual or a remote devices. Bitrise, for example offers a beautiful, visual website, where build steps such as *Gradle Unit Test, Xcode Test for iOS, Xamarin Test Cloud, TestFairy* or *Calabash UI tests* can be added and configured (see figure 2).

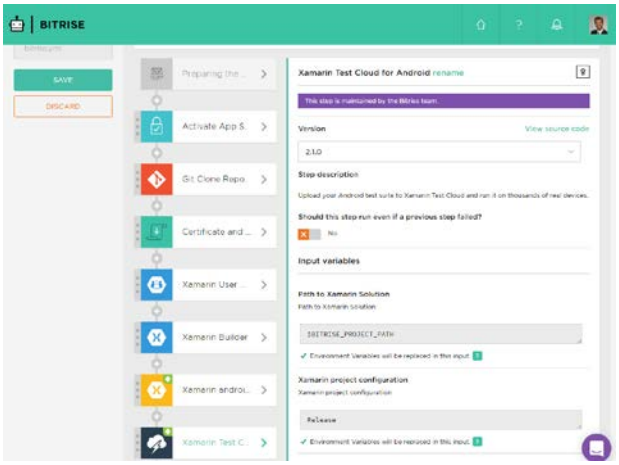


Figure 2: Bitrise website to configure the individual steps of the build

Testing on real devices

No application should ever be released to a larger group of users without being tested at least once by a real human being. But that does not mean that all the tests have to be performed manually, especially not if you consider the wide range of target devices that your application might run on if you release it publicly into an app store. So the testers should primarily focus on the changed bits and verify that the overall application runs smoothly. For all other parts, automated UI-tests should dig deep into the application and verify that all existing workflows produce the expected results on all devices.

If you just want to verify that all visual elements are visible on the screen and the interactions behave as expected when using different locales with various resolutions and platform versions, you can setup your CI to start a couple of different emulators with the specific configurations and run the tests on all of them. This leaves the question open, why one should test on real devices at all? The answer is that some combinations of vendor, model and operating system can show unexpected behavior such as different image rotations when accessing the camera. You can only be sure that your app runs fine, if you have tested it on a wide range of devices.

For testing an application on real devices, a couple of cloud-services are available that are usually payed per usage. Examples of such services are Bitbar Testdroid, Google Firebase, Amazon Web Service Device Farm, Xamarin Test Cloud or TestObject. These services run your application, execute your test-code on the selected number of real devices and usually provide extensive information on the results, such as detailed diagnostics, screenshots or entire videos of the test-run (see figure 3).

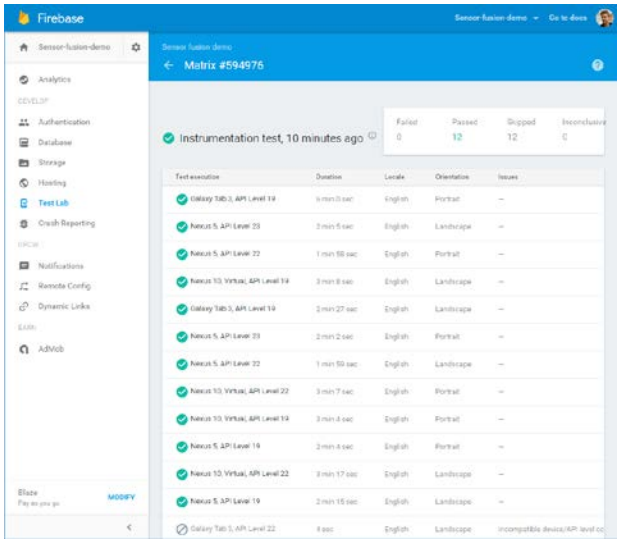


Figure 3: Summary page of a test run on various physical devices with Google Firebase TestLab

One might feel tempted to setup his own farm of physical devices when running an on-premise Continuous Integration server. I would highly discourage this, as setting up such a farm is non-trivial, way more expensive than using cloud-services and one will have to deal with operational issues such as overheated batteries or system-updates that distract you from the actual testing of your application. Note that some cloud services offer a better integration into other tools but for simply automating UI-tests, they are pretty much the same and switching between providers is very easy.

3 – Deploy your app automatically

Once you are happy with you app and gained some confidence from your automated and manual tests, it is time to release your application into an actual app store.

For publishing apps into the Google Play store, Google offers an API that allows you to upload builds (apk-files) to the Google Developer Console, publish it to beta-users or roll-out a new version of your application to a specified percentage of your user-base.

Having this API, one is able to specify release environments (e.g. alpha, beta, production) inside of your CI and push binaries through those environments with a few clicks. Visual Studio Team Services for example offers release definitions that request the consent of dedicated authorities before advancing from one environment to the next one (see figure 4).

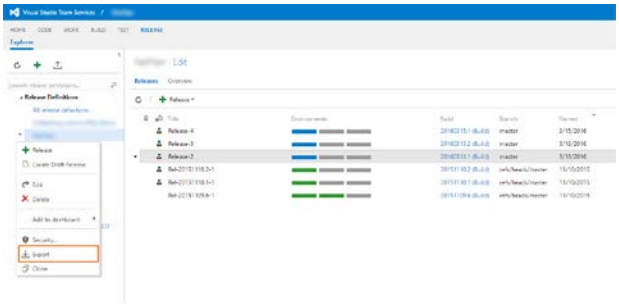


Figure 4: Visual Studio Team Service with multiple release definitions for different environments.

For publishing iOS applications, the process is a bit more elaborate, as Apple demands updated screenshots for each version amongst other information. But Apple provides an API too that allows to upload builds (ipa/pkg-files) to iTunes Connect. A great tool for automatically preparing and uploading the application is called *fastlane*. It is a set of command-line tools that automate the whole process into a few commands that can run automatically on the CI-server.

With these tools, one could even automate the entire process in a way, that each commit onto the master-branch triggers a full release of the application into production if all previous steps completed successfully such as automated tests. This makes sense, if development follows the *GitFlow* branching model, initially proposed by Vincent Driessen and nowadays adopted by many Git clients, where the primary development is done on the develop-branch and each commit onto the master-branch is equivalent to a release.

Monitoring your app

Having an app finally released to the public is usually not where development stops. New features need to be implemented and reported bugs have to be fixed as soon as possible. But how to know that your application has bugs, apart from the fact that *there is no bug-free software*? The answer is by monitoring your application.

A well-known tool for monitoring a mobile application is HockeyApp that was acquired by Microsoft in 2014. HockeApp provides an SDK that can be integrated into any mobile application to automatically send usage statistics and crash-reports. It also makes it easy to distribute beta versions to selected testers and allows them to send feedback to the developers.

Crash reports with a full stack-trace can be vital when tracing bugs in your application, as they contain more insights into the application than any user could formulate verbally (see figure 5).

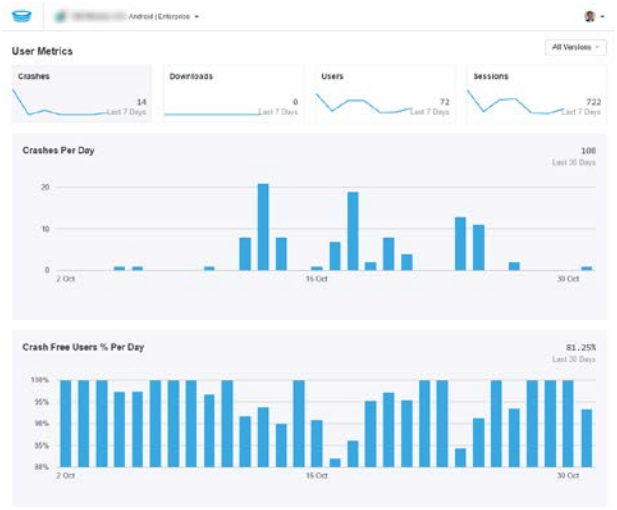


Figure 5: Crash statistics collected by HockeyApp

Conclusion

Continuous Integration and Continuous Delivery are powerful processes that should be used in all software development projects. There are some unique challenges when applying the process to mobile development. But recent advances in the tooling and with cloud-services, offering CI platforms as a service, have made it really easy for all developers to do Continuous Delivery. Even for difficult operations such as signing or testing an application on real devices, there are production-ready solutions that are affordable and work well.

Bibliography

Duvall, Paul et. al, *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007

Humble, Jez et. al, *Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation*, Addison-Wesley, 2010

Driessen, Vincent, A successful Git branching model, 2010:

http://nvie.com/posts/a-successful-git-branching-model/

Cloud-services that facilitate **Continuous Integration and Continuous Delivery** for mobile app:

https://www.bitrise.io/

https://greenhouseci.com/

https://www.visualstudio.com/team-services/

https://travis-ci.org/

An extensive guide to **signing iOS applications**:

https://codesigning.guide/

Samples for **automated Android testing**:

https://github.com/googlesamples/android-testing

Build-automation toolset for iOS and Android:

https://fastlane.tools/

Author

Alexander Pacha

Alexander Pacha is a Software Engineer at Zühlke Engineering and responsible for the development of mobile and C# enterprise applications. He is also a trainer for Clean Code.

alexander.pacha@zuehlke.com, @PachaAlexander

Going DevOps

Questions to Answer at the Beginning of the Journey

Do you intend to introduce DevOps in your company? Do you know how to start? Do you know where to go?

We started the journey to DevOps. And we would like to share our experience to help you take the right turns on your way.

We are going to ask some questions which you should find answers to when you begin and while you are on your way to DevOps.

Where and Why We Started the Journey

Fujitsu Enabling Software Technology GmbH (Fujitsu EST) is a subsidiary of Fujitsu Limited, based in Munich. Our mission is to develop innovative middleware and applications for public, private, and hybrid cloud.

On the one hand, this context requires a high degree of agility and flexibility with short delivery cycles. On the other hand, the strict and substantial quality standards of Fujitsu need to be ensured and maintained in the software products.

In 2009, we started to use agile methods (SCRUM) for the development of our products. We continuously improved the development processes and environment as well as the product management and release processes in the following years with the result that releases fulfilling the quality requirements could be made available basically every month.

Apart from the general goal to further improve our processes and environment, the following events caused us to take the DevOps approach:

With our products, we got increasingly involved in open source and OpenStack activities and projects [1]. This resulted in requirements for even higher agility, more frequent deliveries, and easily deployable artifacts such as Docker containers.

Development of a new product, Fujitsu Cloud Service PICCO [2], was started. We intended to operate PICCO on our own as a service in the cloud. Operation topics thus became much more important.

Since the development of PICCO started from scratch, there was a high degree of flexibility as to the product itself (architecture, features), its artifacts, delivery, and operation environment, and the team of people taking care for it. Bringing together development and operations and covering the lifecycle of a product including its productive use as a service in the cloud became new opportunities and challenges.

With more and more companies talking about DevOps, Fujitsu as a modern IT company also started activities in this area. Fujitsu EST intended to contribute to these activities with an approach based on practical experience.

On our way to DevOps, we were supported by experts from the Fraunhofer IESE. They helped us with their knowledge from other projects and with state-of-the-art and state-of-the-practice analyses. And they guided us by asking many questions and helping us find the right answers.

The next sections present the most important ones of these questions and some hints on how to deal with them.

What Does DevOps Mean to You?

DevOps is an artificial word, composed of Development and Operations. The main intention of DevOps is to remove the barriers between development and operations.

That much is clear. However, if you ask 10 people what DevOps means for them in more detail, you will probably get 15 different answers. They range from “fast time to market” and “full deployment pipeline” through “benefitting from synergies” and “shared responsibilities” of development and operations to “quick customer feedback”, “trying to integrate different mindsets”, or even “pure marketing”.

When you look at the definitions and interpretations of DevOps at different companies or in the Internet, you will come up with a similar variety. An interesting observation here is that many of the definitions do not mention any business aspects or the customer. One of the rare exceptions is the definition by IBM, which also summarizes our understanding at Fujitsu EST:

> *DevOps is an approach **based on lean and agile principles** in which business owners and the development, operations, and quality assurance departments collaborate to **deliver software in a continuous manner** that enables the business to more quickly seize market opportunities and **reduce the time to include customer feedback**.*

No matter which definition you go for: It is vital that everybody taking part in your DevOps activities has the same understanding and moves in the same direction. If this is not the case in the beginning, make it one of your first steps to find such a common understanding.

Not doing so will cause unnecessary and time-consuming distractions later.

What Are Your Goals of Introducing DevOps?

The goals of introducing DevOps vary in the same way as the definitions. While one company may want to release applications more quickly and more often, another one may want to use DevOps to improve the quality and performance of its applications. Each goal requires specific measures and procedures in order to be achieved.

When starting to define your goals, it is no harm to think big. Collect all your dreams and wishes about DevOps, everything you would like to take into account.

Then, look at the constraints and anti-patterns. For example, the structure of your organization or the architecture or operation environment of your products may make it difficult to introduce DevOps. Explicitly state the things you do not want to take into consideration.

By bringing together your wish list and the constraints, you can define realistic goals for a reasonable time frame.

At Fujitsu EST, we defined the following goals:

Implement and use a DevOps environment, including the required tooling and processes. Start with one project and implement a complete deployment pipeline within about half a year.

Use our own products and solutions as much as possible, for example, container technology based on Docker and Kubernetes, or our monitoring solution based on OpenStack/Monasca.

Provide quantitative evidence (metrics, KPIs) of how DevOps affects the performance and output of the teams, and that DevOps does not have a negative impact on the quality of the products.

Collect practical experience, spread it to other projects and organizations, and become an incubator for the Fujitsu group regarding DevOps practices.

What Is Your Strategy? Who Will Do What?

The introduction of DevOps affects the following areas:

Processes: the activities performed during the development, delivery, and operation of your software

Tools: the software, services, and platforms used in the overall process

Organization: your company, its internal structure, team characteristics, decision making processes, culture, and mindset

Architecture: the blueprint of the software and its decomposition, the assignment to teams, and the integration of the components

In order to define your strategy of introducing DevOps, start by analyzing these areas to see what you have got. Then, determine what is missing and what needs to be changed with respect to your goals.

It is important to take a systemic approach considering everything from company goals and business models to the required investments, changes, and time.

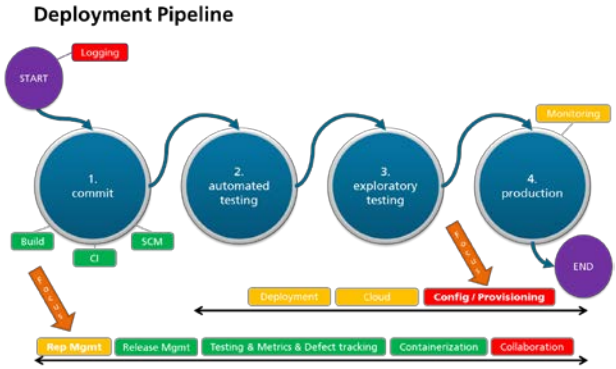
You also need to understand your overall development and operations context with the following dimensions: DevOps activities, responsibilities, products, and customers.

For example, you may want to carry out all DevOps activities by yourself or outsource some of them, such as the operation of the infrastructure. This may differ for individual products and customers.

A conceptual framework that may help you define your DevOps strategy is CALMS, which stands for **C**ulture, **A**utomation, **L**ean, **M**etrics, and **S**haring [3].

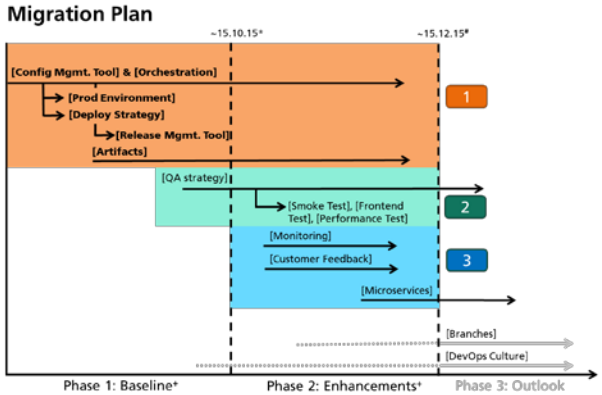
The introduction of DevOps typically involves many changes and big changes, which cannot be introduced in a big bang manner, but require an incremental approach. Start small, proceed step by step, and act according to your requirements and priorities.

At Fujitsu EST, we started the introduction of DevOps with the PICCO project. One of our goals being the implementation of a complete deployment pipeline, we looked at what we had and where we needed to improve:



Processes and tools for development were already well defined and in use, so we focused on operations topics such as configuration management and deployment.

Based on our analysis, our requirements, and the business cases for PICCO, we defined a plan with activities and topics required to create a DevOps environment and address further topics like microservice architecture and quality assurance strategies:



At the beginning, not every aspect was defined in full detail. Based on existing best practices and experiences from other environments and companies, we started changing the processes, selected the tools, and implemented them. Further adaptations followed, based on the experience we made in the steps before.

What Are the Effects of Your DevOps Activities?

The basic concepts of DevOps include continuous integration and delivery as well as **continuous monitoring and feedback**. The goal is to detect issues early and adjust as quickly as possible.

Apply these principles also when introducing DevOps:

Check the effects of each step you take with respect to your goals and requirements.

If necessary, adjust your strategy and steps, your requirements, or even your goals.

When assessing the effects of your DevOps activities, use quantitative data as much as possible, as they provide clarity and help to avoid misunderstandings. Define metrics and key performance indicators (KPIs) that cover the different areas affected by your activities, for example, processes, organization, and products.

At Fujitsu EST, we selected the following metrics in order to continuously monitor the effects of our DevOps activities:

Category	Metric
Process	Deployment time and frequency
Process	Mean time to recover
People	Developer productivity
Business	Time to market for new feature
Business	Expense

However, DevOps also involves aspects that you cannot measure or express as simple numbers, most important being the cultural aspects. For a successful introduction of DevOps, it is indispensable to assess and address these aspects.

For this purpose, we used a questionnaire which we distributed within our DevOps team as well as to people outside the team including management. We asked for ratings for about 50 questions, covering topics like expectations, attitude, perception, collaboration, and motivation related to DevOps. Based on the results, we derived improvement ideas for our DevOps environment as well as issues we needed to address, for example, the reduction of staff fluctuation.

What We Achieved

About one year after starting our journey to DevOps, we have made good progress with respect to our goals:

Implement and use a DevOps environment: For our PICCO project, we have implemented a stable and motivated DevOps team and a complete deployment pipeline. It takes about 10 minutes from a commit to the deployment and completion of automated tests, about 20 minutes up to the completion of manual tests and deployment in the productive environment in Google Cloud. Features that support DevOps have been implemented in the product itself, such as configuration options and feedback mechanisms. Both within and outside the team, we notice an increasing awareness for operations topics and DevOps as such.

Use our own products and solutions: For deploying PICCO, we apply Docker container technology, profiting from the experience of our Docker/Kubernetes team.

Provide quantitative evidence of the DevOps effects: We have started to implement the DevOps metrics we selected for continuous measurement and evaluation. Additionally, we use our existing KPIs to prove that DevOps does not have negative impacts on the quality of our products.

Collect practical experience, spread it to other projects and organizations: We are continuously assessing and improving our DevOps environment and processes and applying parts of them to other projects.

And we are starting to share our knowledge, for example, by actively participating in this conference and writing this article.

References

[1] Open source project Open Service Catalog Manager (http://www.openservicecatalogmanager.org); contributions to OpenStack/Monasca (http://www.openstack.org) and Kubernetes (http://kubernetes.io)

[2] FUJITSU Cloud Service PICCO –

http://www.cloudservicepicco.com

[3] Definition of CALMS:

http://whatis.techtarget.com/definition/CALMS

Acknowledgments

We would like to thank the people from the Fraunhofer IESE, particularly Taslim Arif and Dr. Frank Elberzhager, who is the co-author of this article and the co-speaker of the presentation with the same title in the practical track.

Author

Inge Süß

At FUJITSU Enabling Software Technology GmbH, Inge Süß is responsible for process management and documentation across all projects. Recently, her focus was on introducing DevOps and gathering the corresponding theoretical knowhow and practical experience.

inge.suess@est.fujits

Axiomatic Semantics in a Nutshell

What Every Software Engineer Should Know on Axiomatic Semantics

Among the most valuable aids in reasoning about 3GL code are embedded assertions that are known to be true when reached by program execution. Not only can such assertions facilitate intuitive comprehension, but they are also amenable to precise calculation based on the program statements they surround. Officially known as "axiomatic semantics", this calculus of assertions dates back to the 1960s but is still hardly known among software engineers today. At the same time, however, even experienced professionals routinely fail on small problems like constructing a correct binary search! This paper tackles both issues by teaching a practical style of assertional reasoning and applying it to derive a convincing binary-search solution.

1. BINARY SEARCH – THE PROBLEM

Consider the following little programming task: An array **a** is filled from index **1** through **n** in ascending order. The question is how many of these **n** values are less than a given value **x** (which has the same data type as the elements of **a**). Note that since **a** is sorted, we needn't look at every single element: If an element in the middle is less than **x**, so are all elements before it. If an element in the middle is greater than **x**, so are all elements after it. Thus, our goal is an efficient *binary-search* loop that keeps eliminating half of the unsearched elements until the last element less than **x** has been identified.

Not difficult in principle, this problem has puzzled generations of programmers because getting all the details right is anything but trivial. (Years ago, I used to collect binary searches from workshop participants but eventually stopped after the 17th incorrect solution.) Obviously there are programming tasks for which our "natural" way of thinking is not sufficient – binary search being one striking example. So let's now turn from *operational semantics* ("what the computer does") to *axiomatic semantics* ("what we know at which point"), and the first step in this direction is to express the desired outcome as a simple *assertion*. If **i** is the number of elements less than **x**, then **a(i)** is the last element less than **x** and **a(i+1)** is the first element at least as big as **x**:

[…]
{ a (1..i) < x ≤ a (i+1..n) }

The placeholder **[…]** stands for the program to be designed, and the predicate within curly brackets is the assertion we want to hold immediately after. Note the possibility that **i** equals **0** or **n**, in which case our assertion contains the empty sub-array **a(1..0)** or **a(n+1..n)**, respectively. But this is not a problem, because *anything* is true for elements that don't actually exist. In the extreme case **n = 0**, the solution is **i = 0** with **a(1..0) < x ≤ a(1..0)**.

What's left is to find suitable program statements and intermediate assertions, such that the specified final assertion is provably established. In order to do so, however, we first need the knowledge how to properly "play the game" of assertional reasoning. This is the topic of the following section.

2. THE RULES OF THE GAME

During the early history of axiomatic semantics, a variety of theoretical styles have been developed. Peter Naur analyzed algorithms using snapshots [1]; Robert Floyd attached assertions to flowcharts [2]; Tony Hoare worked with triples of the form **{precondition} program {postcondition}** [3]; and Edsger W. Dijkstra focused on weakest preconditions for given postconditions [4].

However, from a practical software-engineering viewpoint, it turns out that the formal differences between these theories are immaterial. Instead, all we need is a well-balanced mix of source code and assertions (sometimes called *proof outline* or *annotated program*), where every programming construct is locally in balance with the assertions next to it. So with this general plan in mind, let us now have a look at different constructs and their balancing rules.

2.1. How to jump over assignments

Let **v** be an arbitrary program variable and **τ** a valid programming-language term whose data type is compatible with **v**. Then the assignment **v := τ;** means that **τ** is evaluated in the current run-time environment, and the result is stored as the new value of **v**. While this *operational* meaning is of course basic programming knowledge, it's much less obvious how it translates to an equivalent *assertional* meaning. In other words, if **P** and **Q** are predicates over program variables, how can we know if the following fragment balances?

{ P }
v := τ;
{ Q }

Although it's clear that the relationship between **P** and **Q** must have to do with variable **v** and term **τ**, the exact rule which the pioneers of axiomatic semantics found out seems hardly intuitive to anyone who sees it for the first time. The trick is that in contrast to program execution, which flows downward from **P** to **Q**, assertions have to be calculated just the other way round, i.e. we start at **Q** and replace **v** by **τ** as we "jump upward" over the assignment:

{ [v := τ] Q }
v := τ;
{ Q }

Note that the square-bracket notation means substitution, and that the resultant predicate **[v:=τ]Q** is always the least restrictive (or *weakest*) **P** satisfying **{P} v := τ; {Q}**. Since this also happens to be the "best" **P** for practical purposes, we shall not explore alternative **P**s at this point but illustrate the rule just learned with a little example.

2.1.1. Swapping with temporary variable

Suppose we have two assignment-compatible variables **a** and **b** and want to exchange their values. Unlike binary search, this problem is so simple that operational reasoning is perfectly sufficient. All we need to do is save the old value of **a** in a temporary variable, copy **b** to **a**, and then copy the saved value from the temporary variable to **b**:

t := a;
a := b;
b := t;

As this program consists only of assignments, we should already be able to calculate assertions for it. Let's refer to the original values of **a** and **b** by what's called *rigid variables* **A** and **B** (written in uppercase to distinguish them from program variables). Then the goal of our program can be captured by the final assertion **b=A ∧ a=B** [**∧** meaning "and"], and by repeated calculation from bottom to top, we eventually get a fully annotated program:

t := a;

a := b;

b := t;
{ b = A ∧ a = B }

	{ a = A ∧ b = B }
t := a;	t := a;
	{ t = A ∧ b = B }
a := b;	a := b;
{ t = A ∧ a = B }	{ t = A ∧ a = B }
b := t;	b := t;
{ b = A ∧ a = B }	{ b = A ∧ a = B }

Note how each step of this calculation yields a new assertion by performing a correct "upward jump". And if we now look at *precondition* **a=A ∧ b=B** and *postcondition* **b=A ∧ a=B**, we see that the net effect is indeed a variable swap.

2.1.2. From verification to construction

The previous example shows a typical mix of operational and assertional reasoning: First, operational semantics was used to "guess" the correct program, and then axiomatic semantics was used to verify the result. But wouldn't it be interesting to watch the whole program emerge from assertional reasoning alone? In other words, can we construct the annotated program as a single entity, yielding the correctness proof as a by-product of the construction process?

From a purely axiomatic viewpoint, we are looking for a *predicate transformer* that turns postcondition **b=A ∧ a=B** into precondition **a=A ∧ b=B**. However, if we try one of the substitutions **[b := a]** or **[a := b]**, we obtain:

{ a = A ∧ a = B }	i.e. a = A = B
b := a;	
{ b = A ∧ a = B }	
{ b = A ∧ b = B }	i.e. b = A = B
a := b;	
{ b = A ∧ a = B }	

Of course, both fragments have to be read from bottom to top again, and we see that the assertions on top can only be true if **A = B** (meaning that the two variables to be swapped have the same original value). But since we want a program that also works for **A ≠ B**, we must change the right-hand side of the assignment, e.g. to a new variable **t**:

{ t = A ∧ a = B }	Now we can safely apply
b := t;	**[a := b]** without running
{ b = A ∧ a = B }	into **A = B** again!
{ t = A ∧ b = B }	This looks almost like the
a := b;	precondition already; we
{ t = A ∧ a = B }	just need to apply **[t := a]**
	to finish our construction.

2.1.3. Swapping without temporary variable

Time for a more advanced exercise: Can we get rid of the temporary variable and still swap **a** and **b** correctly (even if **a ≠ b**)? Obviously not if the right-hand side of the last assignment is just **a** or **b** – as we saw in the previous section. So why not try the composite term **a + b**, assuming e.g. that **a** and **b** are integer variables?

{ b = A ∧ a + b = B }
a := a + b;
{ b = A ∧ a = B }

Note how closely the new predicate **a + b = B** resembles **b = B**. All we need to do is "subtract" **a** somehow, in the simplest case by one of the substitutions **[a := a – a]** or **[b := b – a]**. The first option boils down to **[a := 0]**, trapping us in the undesirable predicate **b = A = B** again:

{ b = A ∧ 0 + b = B }
a := 0;
{ b = A ∧ a + b = B }

So maybe subtracting from **b** is more promising?

{ b – a = A ∧ a + (b – a) = B }
b := b – a;
{ b = A ∧ a + b = B }

Looks indeed much better, especially because **a + (b – a) = b**. And if we introduce a **NULL** statement that does nothing, we can formally continue with:

{ b – a = A ∧ b = B }
NULL;
{ b – a = A ∧ a + (b – a) = B }

Now we shouldn't touch **b** any more, in order to preserve **b = B**. Hence, we'll have to change **a** to achieve the missing transformation **b – a → a**. But what term **τ** shall we assign to **a**? Let's see if we can calculate the answer:

{ b – τ = A }
a := τ;
{ b – a = A }

So if we want to have **a=A** on top, we only need to solve **b – τ = a** for **τ**:

b – τ = a | + τ
b = a + τ | – a
b – a = τ

In other words, our last missing substitution is **[a := b – a]**:

{ a = A ∧ b = B }
NULL;
{ b – (b – a) = A ∧ b = B }
a := b – a;
{ b – a = A ∧ b = B }

Thus, the complete program looks as follows:

{ a = A ∧ b = B }
NULL;
{ b – (b – a) = A ∧ b = B }
a := b – a;
{ b – a = A ∧ b = B }
NULL;
{ b – a = A ∧ a + (b – a) = B }
b := b – a;
{ b = A ∧ a + b = B }
a := a + b;
{ b = A ∧ a = B }

2.2. How to rewrite assertions

For the sake of completeness, let us also try to characterize the **NULL** statement used in the previous construction. Its operational meaning is simply to leave all variables unchanged (i.e. to "do nothing"), and from an assertional viewpoint we can say that a **NULL** statement is in balance with two equivalent assertions around it. Formally this gives rise to a *conditional* balancing rule [with a bar under the condition and ⇔ meaning "equivalent"]:

P ⇔ Q
———
{ P }
NULL;
{ Q }

Practically speaking, we can always "rewrite" an assertion to some other equivalent formula and put a **NULL;** in between (or just empty space where permitted by programming-language syntax).

2.2.1. Alternative swapping solutions

You may wonder if we have just been lucky that the composite term **a + b** in 2.1.3. worked out so nicely. What if we had started with integer *subtraction* instead of addition? Well, let's see:

{ a = A ∧ b = B }	{ a = A ∧ b = B }
{ (a + b) – b = A ∧ b = B }	{ (a – b) + b = A ∧ b = B }
a := a + b;	a := a – b;
{ a – b = A ∧ b = B }	{ a + b = A ∧ b = B }
{ a – b = A ∧ a – (a–b) = B }	{ a + b = A ∧ (a+b) – a = B }
b := a – b;	b := a + b;
{ b = A ∧ a – b = B }	{ b = A ∧ b – a = B }
a := a – b;	a := b – a;
{ b = A ∧ a = B }	{ b = A ∧ a = B }

Apparently it doesn't matter *how* **a** and **b** are combined in the last statement (as long as the total amount of information is preserved). But once that decision is made, the rest is dictated by the first goal **b = B** and the second goal **a = A**. Also note the four "invisible **NULL** statements" at the points where assertions have been rewritten.

2.3. How to split into branches

We shall now turn from assignment sequences to programs whose execution path depends on run-time conditions. Let **β** be a Boolean programming-language term without any side-effect on program variables. Then the **IF** construct

```
IF β THEN
        […]
ELSE
        […]
END IF;
```

operationally means that **β** is evaluated in the current run-time environment, and depending on the outcome either the **THEN** branch is executed (viz. if the evaluation has yielded **TRUE**) – or the **ELSE** branch is executed (if **FALSE**).

Like with assignment and **NULL** statements, we would like to know when the following **IF** fragment balances:

```
{ P }
IF β THEN
        […]
ELSE
        […]
END IF;
{ Q }
```

Clearly, this will depend on balancing conditions within the two branches. If **P** holds before the **IF**, it will still hold after the **THEN** or **ELSE** (thanks to **β** being free of side-effects). In addition, we can assert **β** at the beginning of the **THEN** branch and **¬β** at the beginning of the **ELSE** branch [¬ meaning "not"]. And in order to guarantee **Q** after the **END IF**, we need to guarantee it at the end of either branch:

```
{ P }
IF β THEN
        { P ∧ β }
        […]
        { Q }
ELSE
        { P ∧ ¬β }
        […]
        { Q }
END IF;
{ Q }
```

Note that unlike the balancing rules we have seen before, this new rule is not "self-contained" in that it contains unknown parts indicated by **[…]**. However, these unknown parts are constrained by embedded assertions **{ P ∧ β }**, **{ P ∧ ¬β }**, and **{ Q }** – so if they both balance locally, the rule makes sure that the fragment is in balance as a whole.

2.3.1. Swapping once again

Our next example addresses a potential problem with the last swapping programs based on integer arithmetic. Everything would be fine within the set **Z** of mathematical integers, which is closed under addition and subtraction. However, a real-world variable **v** can only represent a finite subset of **Z** – typically within the limits **$-L \le v < L$**, where **L** is some large power of **2**. So in order to "survive" the first assignment, we need **$-L \le a \pm b < L$** in addition:

```
{ a = A ∧ b = B ∧ –L ≤ a+b < L }
a := a + b;
b := a – b;
a := a – b;
{ b = A ∧ a = B }
```

```
{ a = A ∧ b = B ∧ –L ≤ a–b < L }
a := a – b;
b := a + b;
a := b – a;
{ b = A ∧ a = B }
```

Note that the second and the third assignment are always safe: The postcondition tells us that **b** becomes **A** and **a** becomes **B**, both of which are valid machine integers to begin with. (The formal safety proofs are left as an exercise.)

We see that neither program works in all cases: If **a** and **b** have the *same sign* (plus or minus) and their total absolute value is too big, we can't compute **a + b**. On the other hand, if they have *different signs* and are too far apart, we can't compute **a – b**. But within the same sign we can always *subtract*, and across different signs we can always *add* (even if we put the "sign-less" **0** together with the positive integers to reduce the number of combinations to be analyzed):

$$\frac{-L \le a < 0 \qquad -L \le b < 0,\ \text{i.e. } 0 < -b \le L}{-L+0 < a-b < 0+L} \qquad \frac{-L \le a < 0 \qquad 0 \le b < L}{-L+0 \le a+b < 0+L}$$

$$\frac{0 \le a < L \qquad -L \le b < 0}{0-L \le a+b < L+0} \qquad \frac{0 \le a < L \qquad 0 \le b < L,\ \text{i.e. } -L < -b \le 0}{0-L < a-b < L+0}$$

Now we have all the ingredients for a robust integer swapper without temporary variable:

```
{ a = A ∧ b = B }
IF (a < 0) = (b < 0) THEN
        { a = A ∧ b = B ∧ (a < 0) = (b < 0) } i.e.
        both are negative or both are non-negative
        { a = A ∧ b = B ∧ –L < a–b < L }
        { a = A ∧ b = B ∧ –L ≤ a–b < L }
        a := a – b;
        b := a + b;
        a := b – a;
        { b = A ∧ a = B }
ELSE
        { a = A ∧ b = B ∧ (a < 0) ≠ (b < 0) } i.e.
        one is negative and the other non-negative
        { a = A ∧ b = B ∧ –L ≤ a+b < L }
        a := a + b;
        b := a – b;
        a := a – b;
        { b = A ∧ a = B }
END IF;
{ b = A ∧ a = B }
```

2.4. How to strengthen and weaken

The attentive reader may have noticed that the (invisible) **NULL** statements in the previous example are *not surrounded by equivalent assertions* – as would have been required by our balancing rule from section 2.2. Although we do know that **(a < 0) = (b < 0)** implies **–L < a–b < L**, for example, the opposite direction doesn't necessarily hold, as the counter-example **a = –1 ∧ b = 0** illustrates. The same is true for the next **NULL** transformation, where "<" (above) turns into "≤" (below) to achieve a seamless connection to the sub-fragment within the **THEN** branch.

So rather than being equivalent, the upper assertion is always more restrictive (or *stronger*), and the lower assertion is more relaxed (or *weaker*) in this example. Nevertheless, such a **NULL** fragment is still in balance according to theory – and we can generalize the double-headed *equivalence arrow* [⇔] to a single-headed *implication arrow* [⇒]:

```
P ⇒ Q
-----
{ P }
NULL;
{ Q }
```

2.4.1. The invisible branch

A final warning about the **IF** construct is in order. Many programming languages offer a variation where the **ELSE** branch is omitted, so you might be tempted to believe that the **ELSE** branch in the balancing rule is optional as well:

```
                        { P }
IF β THEN               IF β THEN
                                { P ∧ β }
        […]                     […]
                                { Q }
END IF;                 END IF;
                        { Q }
```

This reasoning is flawed, however, because the operational meaning of above construct is to execute the **THEN** branch if **β** holds and to "do nothing" if **¬β** holds. But "doing nothing" is a **NULL** statement – so the correct rule is:

```
                        { P }
IF β THEN               IF β THEN
                                { P ∧ β }
        […]                     […]
                                { Q }
ELSE                    ELSE
                                { P ∧ ¬β }
        NULL;                   NULL;
                                { Q }
END IF;                 END IF;
                        { Q }
```

Of course, the **ELSE** branch is subject to the standard **NULL** rule, in this case requiring **P ∧ ¬β ⇒ Q**. And this is exactly the pitfall of an "invisible" **ELSE** branch: The **THEN** branch may be perfectly in balance, but if you forget to balance the **ELSE** branch too (which can easily happen if you don't see it), you cannot assert **Q** after the **END IF**.

2.5. How to balance a loop

The last construct we want to study is a loop with a Boolean term **β** evaluated at the very beginning. If **TRUE**, the **LOOP** body is executed, followed by a re-execution of the whole construct. If **FALSE**, execution continues after the **END LOOP**. Thus, if **P** holds initially and is kept *invariant* by the **LOOP** body, we can assert **P ∧ ¬β** at the end:

```
                        { P }
WHILE β LOOP            WHILE β LOOP
                                { P ∧ β }
        […]                     […]
                                { P }
END LOOP;               END LOOP;
                        { P ∧ ¬β }
```

The **LOOP** body of the **WHILE** construct has some similarity with the **THEN** branch of the **IF** construct. Again, we require **β** to be free of side-effects, so if **P** holds before the **WHILE**, it will still hold inside the

loop – together with **β**, because otherwise the **LOOP** body wouldn't have been entered. However, instead of some other postcondition **Q**, the **LOOP** body has to re-establish the same invariant **P** at the end, as a precondition for the ensuing re-execution.

2.5.1. Safety and progress

While programs composed of only assignment, **NULL**, and **IF** constructs are guaranteed to finish in a finite time, the **WHILE** construct just introduced may loop forever if the **LOOP** body doesn't eventually force **β** to **FALSE**. The *safety property* **{ P ∧ β } […] { P }** is clearly not enough, as the drastic example **{ P ∧ β } NULL; { P }** illustrates. Such a **LOOP** body would always satisfy the previous rule (because **P ∧ β ⇒ P**), but there is no hope that **β** will ever change from **TRUE** to **FALSE**.

On the other hand, we must not insist that the **LOOP** body establish **¬β** immediately when first executed, because it's perfectly OK to need two or more iterations – as long as it's a finite number. What we do need, however, is a gradual progress in the "right direction", which in turn requires a non-Boolean term **μ** (called a *metric*), which gets smaller with every execution of the **LOOP** body.

The metric **μ** needn't necessarily be a programming-language term, but the set of all possible values of **μ** must be what's called *well-founded*, meaning that all decreasing chains (**μ > μ' > μ'' > ...**) are finite. The set **N** of natural numbers is well-founded, for example, whereas the set **Z** of integers is not. Only under the assumption of well-foundedness does the *progress property* **{ μ = M } […] { μ < M }** ensure termination – with the rigid variable **M** capturing the old value of **μ** for a given iteration:

{ P }
WHILE β LOOP
    { P ∧ β ∧ μ = M }
    […]
    { P ∧ μ < M }
END LOOP;
{ P ∧ ¬β }

3. BINARY SEARCH – THE SOLUTION

We are now ready to pick up the binary-search problem from the beginning of this paper. If we look at the desired postcondition **a (1..i) < x ≤ a (i+1..n)**, we see that the two interesting indices **i** and **i+1** have a difference of **1**. This is of course perfect for a *finished* binary search, but we'll certainly need more flexibility while the search is still in progress. So let's generalize from **i < i+1** to **i < j**, with a new variable **j** assuming the role of **i+1**:

P: $0 \le i < j \le n+1 \wedge a(1..i) < x \le a(j..n)$

The simplest initialization is obviously with **i** and **j** as far apart as possible, such that both sub-arrays become empty. And if we then enter a loop with **P** as invariant, we only need to compare **i+1** with **j** in the **WHILE** condition:

i := 0;
j := n + 1;
{ P }
WHILE i+1 ≠ j LOOP
    { P ∧ i+1 ≠ j }
    { P ∧ i+1 < j }
    [...]
    { P }
END LOOP;
{ P ∧ i+1 = j }
{ a (1..i) < x ≤ a (i+1..n) }

Next, we need a well-founded metric to guide us from the initial state with **i** and **j** far apart to the final state with **i** and **j** close together. A reasonable choice is **j – i**, which is always a natural number because **i < j**:

    { j – i = M }
    [...]
    { j – i < M }

What's left is the design of the **LOOP** body, which should bring **i** and **j** closer together under invariance of **P**. The easiest way to achieve this is one of the substitutions **[i := t]** or **[j := t]**, with **t** somewhere in the gap between **i** and **j**:

    { P ∧ i < t < j ∧ a(t) < x }
    { 0 ≤ t < j ≤ n+1 ∧ a (1..t) < x ≤ a (j..n) }
    i := t;
    { P }

    { P ∧ i < t < j ∧ x ≤ a(t) }
    { 0 ≤ i < t ≤ n+1 ∧ a (1..i) < x ≤ a (t..n) }
    j := t;
    { P }

Note that there *has* to be such a gap (thanks to **i+1 < j**), and for efficiency reasons we'll place **t** right in the middle (with arbitrary rounding if the gap has an even length). Also note that **0 ≤ i < t < j ≤ n+1** implies **1 ≤ t ≤ n**, so **a(t)** is always a valid array element. Hence, the following **LOOP** body has all the necessary safety and progress properties:

```
{ P ∧ i+1 < j }                { j – i = M }
t := (i + j) / 2;
{ P ∧ i < t < j }              { j – i = M }

IF a(t) < x THEN
        { P ∧ i < t < j ∧ a(t) < x }   { j – i = M }
        i := t;
        { P }                          { j – i < M }
ELSE
        { P ∧ i < t < j ∧ x ≤ a(t) }   { j – i = M }
        j := t;
        { P }                          { j – i < M }
END IF;

{ P }                          { j – i < M }
```

4. ENGINEERING AND INTELLECTUAL CONTROL

The hallmark of a mature engineering discipline is full intellectual control over the artifacts produced. Long before new buildings or machines actually exist, their relevant properties can be predicted with high reliability – and no serious engineer would skip some of the necessary calculations because they're "too much work" or seem "too difficult". Yet, in the realm of software development, that's what happens all the time: If noticed at all, loss of intellectual control is lightly justified by the fact that the code "will be tested anyway" or is "too tricky to get right".

For a software engineer with axiomatic-semantics knowledge, there are no such excuses. Whenever operational reasoning reaches its limits, elements of assertional reasoning can immediately be added by asking "What do I know at which point?" and embedding the most helpful answers as (semi-)formal assertion comments in the code. Of course, the validity of each assertion must be verified – if necessary by further assertions and detailed balancing. But if properly done, intellectual control is always attainable; even in the case of "tricky" algorithms like binary search.

Acknowledgments

My interest in axiomatic semantics was first sparked by Prof. Werner Kuich at TU Vienna and later magnified by Prof. Edgar Knapp at Purdue University, who has also reviewed drafts of the present paper. The material covered is closely tied to my long-term workshop series at Scientific Games Vienna, whose management team I thank for making these workshops possible. Credit also goes to the numerous workshop participants, the interaction with whom has sharpened my view of assertional reasoning as a key to intellectual control in software engineering.

References

[1]
P. Naur. *Proof of Algorithms by General Snapshots.* BIT, **6**(4):310–316, 1966.

[2]
R. W. Floyd. *Assigning Meanings to Programs.* Proceedings of Symposia in Applied Mathematics, **19**:19–32, 1967.

[3]
C. A. R. Hoare. *An Axiomatic Basis for Computer Programming.* Communications of the ACM, **12**(10):576–583, 1969.

[4]
E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

Author

Dipl.-Ing. Herwig Egghart
Cleanroom Software Engineer
Scientific Games International
egghart@scigames.at

Testumgebungen auf einen Klick

Zeitgemäßes Testumgebungsmanagement als Herausforderung und Lösung

Eine neue Softwarelieferung steht an - und somit ein Regressionstest. Vorher soll noch das Betriebssystem der Testserver einem Update unterzogen werden und das wichtige Drittsystem ist mal wieder nicht verfügbar.

Die Hardware ist seit Monaten zu schwach um alle Systeme in unterschiedlichen Versionen parallel zu betreiben.

Der Wunsch nach vielfältigen Testumgebungen ist seit langer Zeit genau DAS – ein Wunsch.

Diese, und weitere Herausforderungen im Bereich Testumgebungs- bzw. Testinfrastrukturmanagement, gilt es zügig und mit Mehrwert für die gesamte IT-Organisation zu lösen. In diesem Beitrag widmen wir uns daher den Folgenden Fragestellungen:

- Welche Möglichkeiten gibt es um Testumgebungen „auf Knopfdruck“ bereit zu stellen?
- Welche organisatorischen Veränderungen gehen damit einher?
- Die Cloud als Variante für modernes Hosting: Welche Themen müssen dabei berücksichtigt werden?

Ziel ist es, dass Sie als Leser ein Gesamtbild erkennen, wie all diese Aspekte ineinandergreifen, um die Herausforderung Testumgebungsmanagement als Lösung einzusetzen.

Sehen Sie diesen Artikel als Denkanstoß, diesem Thema beim nächsten Quartalsmeeting einen eigenen Punkt auf der Agenda einzuräumen.

Testumgebungen auf einen Klick – Die Voraussetzung

Grundvoraussetzung: Testumgebungen müssen automatisch deployed werden können. Dafür gilt es die manuellen Schritte, die dafür notwendig sind, zu eliminieren. Im Grunde soll es nur zwei manuelle Schritte geben:

- Die Auswahl der Version von dem zu liefernden Software-Artefakt (z.B. eine App oder ein Service)
- Die Auswahl der Umgebung, in welche dieses Artefakt geliefert werden soll

Alles andere erfolgt im Idealfall automatisiert. Die Anweisungen, wie die Installation und Konfiguration der Testumgebung erfolgt, muss dafür als Code vorliegen.

Was bedeutet Infrastructure as Code (IaC)?

Die Bedeutung von IaC kann sich je nach Technologie stark unterscheiden. Das Wichtigste ist aber, dass jeglicher Code unter Versionskontrolle steht und somit versioniert ist.

In vielen Organisationen wird bereits IaC genutzt, nämlich in Form von Shell-, Batch-, oder Powershell-Scripts. Diese werden jedoch rasch zu komplex und sind damit nur unter großem Aufwand wartbar.

Wie eine moderne Herangehensweise an IaC aussehen kann, wird in diesem Artikel mittels der Ansätze bzw. Technologien von Puppet und Docker beschrieben. Diese sind zwar unterschiedlich, schließen sich aber gegenseitig nicht aus und können durchaus sinnvoll kombiniert werden.

Puppet

Puppet ist ein auf Ruby basierendes Framework. Es ermöglicht eine Software samt notwendiger Umgebung automatisch bereitzustellen, zu aktualisieren und zu überwachen.

Dabei wird der gewünschte Zustand in einer eigenen Domain-Specific-Language (DSL) beschrieben und Puppet kümmert sich darum, dass dieser Zustand mit allen Abhängigkeiten erreicht wird. Dazu muss als erstes Puppet auf der Zielplattform installiert sein. Puppet verwendet wiederverwendbare Module, wobei sich die Module um bestimmte Aufgaben kümmern (z.B. Installation und Konfiguration eines Apache Webservers oder einer Datenbank). Es können eigene (custom) Module geschrieben werden oder auch auf tausende von der Community bereitgestellte Module aus der sogenannten „Forge“ zurückgegriffen werden.

Ausgangspunkt ist meist das site.pp Manifest, das z.B. so aussehen kann:

```
# /etc/puppetlabs/puppet/manifests/site.pp
node 'sut.example.com' {
  include my-app
```

```
}
node 'db1.example.com' {
  include postgres
}
```

Dieses File beschreibt, dass auf der Maschine mit dem Domainnamen „sut.example.com“ die Klasse my-app angewendet werden soll. Diese Klasse könnte sich z.B. in einem Custom-Modul befinden in welchem beschrieben ist, wie my-app installiert und konfiguriert wird und welche Abhängigkeiten my-app hat. Dies könnte z.B. so aussehen:

```
class my-app (String $version = '1.8.2') {
  nginx {'nginx':
    ensure => 1.9.15
  }
  mybase::package {"Install my-app $version":
    package_name    => my-app,
    package_version => $version,
    destination_folder => /usr/share/nginx/html,
    require  => nginx
  }
  file {'/etc/nginx/nginx.conf':
    ensure  => file,
    owner   => 'httpd',
    content => template('nginx/nginx.conf.erb'),
    require  => nginx
  }
  service {'nginx':
    ensure    => running,
    enable    => true,
    subscribe => File['/etc/nginx/nginx.conf']
  }
}
```

Alle notwendigen Ressourcen wie Files, Services, User, Packages etc. werden mit den Properties aufgelistet und Puppet sorgt dafür, dass diese Ressourcen (auch in der richtigen Reihenfolge) auf dem Zielrechner vorhanden sind.

Im Beispiel oben benötigt my-app nginx als Webserver. Die Installation erfolgt, indem mybase::package verwendet wird. Diese Klasse „weiß“ alles Notwendige, um ein Package zu installieren - sie kann innerhalb der Organisation Packages aus dem Binary Repository herunterladen und diese installieren (z.B. im destination_folder entpacken).

Es können auch verschieden Abhängigkeiten beschrieben werden. „Require“ beschreibt, dass eine bestimmte Ressource benötigt wird, bevor eine andere angewendet wird (Zuerst nginx installieren, bevor my-app installiert wird.) Subscribe beschreibt, dass das Service nginx neu gestartet werden muss, falls sich /etc/nginx/nginx.conf ändert.

Weitere Konzepte:

Facter: liefert vorgefertigte oder individuelle Informationen über das System, auf das gerade ein Puppet-Run angewendet wird (z.B. Architektur, Netzwerkkonfiguration, OS Version, Umgebungsvariablen). Diese Facts werden in den Modulen verwendet, um beispielsweise je nach Architektur andere Pfade zu verwenden. Dies erhöht die Wiederverwendbarkeit von Modulen, da diese nicht für jede Zielplattform bzw. Architektur neu geschrieben werden müssen.

Hiera: ist ein Key-Value-Lookup Tool für Konfigurationsdaten. Alle umgebungsspezifischen Daten - wie URLs, Ports, Heap Space, usw. – sollen in den Modulen nicht hardcodiert sein, sondern aus einer Konfigurationsquelle gelesen werden. Dadurch verbessert sich die Flexibilität der verwendeten Module, weil Parameter dadurch dynamisch gesetzt werden können.

Docker

Docker ist im Kern eine Betriebssystemvirtualisierung in standardisierten, übertragbaren Containern, die in einer sogenannten Docker Engine laufen.

Ein Image, welches alle umgebungsunabhängigen Vorbedingungen inkludiert, wird einmal erstellt und in allen Umgebungen wiederverwendet. Von diesem Image ausgehend werden Container gestartet - denen z.B. umgebungsspezifische Parameter übergeben werden können. Das Erzeugen und Starten dieser Container ist extrem schnell, da nur ein Prozess gestartet wird - im Gegensatz zu einer neuen Virtuellen Maschine, bei der ein ganzes Betriebssystem gestartet wird. Deshalb können Container einfach entsorgt und wieder neu erstellt werden.

Hier ein Beispiel wie ein Docker-File aussehen könnte, von dem aus ein Image erstellt wird:

```
FROM nginx:1.9.15
MAINTAINER: devopsteam@anecon.com
ENV MY_APP_VERSION 0.7.3
RUN   wget   http://nexus:8081/content/repositories/my_app/my_app-$MY_APP_VERSION.tar.gz \
 && tar -C /usr/share/nginx/html -xvzf my_app-$MY_APP_VERSION.tar.gz
COPY ./app/
EXPOSE 80
VOLUME /var/log/nginx/
ENTRYPOINT ["/app/docker-entrypoint.sh"]
```

Die FROM Anweisung gibt ein Basisimage vor, in dem nginx in einer definierten Version bereits vorhanden ist. Mit Docker Build wird aus diesem Docker-File ein Image gebildet und in einem Image Store (Image Registry) abspeichert. Über Tags kann auf bestimmte Versionen dieser Images zugegriffen werden.

Um z.B. eine Testumgebung zu erstellen in der my-app und eine Postgres Datenbank verwendet wird, genügt ein yaml-File, das die Infrastruktur beschreibt.

Mit docker-compose können die Container dann gestartet werden:

```
version: '2'
services:
  sut:
    image: my-app:0.7.3
    depends_on:
      - db
    ports:
      - "8080:80"
  db:
    image: postgres:9.4.9
    environment:
      - POSTGRES_PASSWORD: mypostgrespwd
      - POSTGRES_INITDB_ARGS: --data-checksums
    volumes:
      - datavolume:/var/lib/postgresql/data
```

Docker bietet aber noch viel mehr – zum Beispiel ein komplettes Ecosystem und Toolset zur Verwaltung dieser Container. Dieses enthält:

- Build-Umgebung, um Images aus Source-Files (Docker-File) zu erstellen.
- Docker Hub: reichhaltiges Angebot an vorgefertigten Images – sowohl offizielle, als auch von der Community zur Verfügung gestellte.
- docker-compose: Definieren und Starten von Multi-Container Applikationen
- docker-swarm: zur Verwaltung von Docker Clustern (Zusammengefasste Docker-Hosts)
- docker-machine: Provisionierung von Docker Hosts

Unterschiede und Gemeinsamkeiten

Bei Docker wird ein definiertes Image unverändert in verschiedenen Umgebungen wiederverwendet. In einem Image ist immer nur ein Service (z.B. Webserver oder Datenbank) enthalten. Alle notwendigen Ressourcen sind im Image enthalten. Das Starten eines neuen Containers basierend auf diesem Image, ist wie das Starten eines neuen Prozesses.

Zum Vergleich: Bei Puppet wird ein gewünschter Zustand des Systems definiert und dieser auf den Zielplattformen hergestellt. Startpunkt ist dabei z.B. ein „golden Image“ (ein genau spezifiziertes Image, das für die Erstellung weiterer Images verwendet wird).

Mit Docker verglichen, ist Puppet eher für Infrastruktur, die länger erhalten wird, geeignet – beispielsweise Testumgebungen in einer späteren Phase der Entwicklung (System-Integrationsumgebung, Pre-Produktion, Hotfix Umgebung). Die Umgebung wird für jede Stage neu erstellt oder eventuell auch aktualisiert, falls das Provisioning von VMs oder physischer Maschinen zu aufwendig ist.

Bei Docker können neue Container ganz einfach ohne Ressourcenverbrauch aus dem Image wieder erzeugt werden - und werden daher auch einfach nach Ende der Tests gelöscht. Dieses Vorgehen ist gut geeignet für Entwicklungsumgebungen, die häufig beliefert werden und sich daher täglich ändern können. Aus Testsicht eignen sich diese Umgebungen gut für Unit- oder Komponententests.

Eines haben jedoch beide Technologien gemeinsam: Es werden Zielmaschinen benötigt, auf denen ein Deployment durchgeführt werden kann. Für das Hosting von Testumgebungen existieren heutzutage einige Möglichkeiten – die spannendste davon wird im Folgenden beschrieben.

Die Cloud: unendliche Weiten

Vor einigen Jahren noch wenigen Leuten ein Begriff, ist die Cloud heute vor allem im privaten Bereich, wie zum Beispiel vom Smartphone (Dropbox, Evernote) oder TV (Netflix, Amazon Prime), nicht mehr wegzudenken. Im Businessbereich stoßen große Anbieter mit Endanwender-Lösungen in neue Marktbereiche vor, wie beispielsweise Microsoft mit Office365. Abseits von diesen offensichtlichen Einsatzmöglichkeiten, bietet die Cloud aber auch Lösungen für alltägliche Probleme in den Bereichen Softwaretest- und Entwicklung – hier sind mit Hilfe der Cloud neue Wege zu beschreiten.

Ist in diesem Artikel von „Cloud“ die Rede, verstehen wir darunter die Services zur Anmietung von Hardware von großen Anbietern, wie Amazon oder Microsoft. Besonders bei Microsofts Azure Cloud, wird darauf Wert gelegt, gängige Microsofttools (z.B. Visual Studio, Sharepoint oder Team Foundation Server) zu integrieren. Hier liegt der Nutzen für IT-Organisationen ohnehin auf der Hand: versionierte Quellcodeablage, automatisierte Build- und Deploymentprozesse, nahtlose Einbindung des Fehlermanagementsystems, Monitoring der eingesetzten Hardware und transparente Darstellung der anfallenden Kosten.

Amazon Web Services (AWS) wiederum bietet einen großen Marketplace, in dem kostenlos oder für geringe Cent-Beträge, Templates für Maschinenkonfigurationen bezogen werden können. Hier können mit wenigen Mausklicks virtuelle Server mit einer Reihe von vorinstallierten Tools (wie z.B Apacheserver) hochgezogen werden.

Testumgebungshosting heute und morgen

Im Vergleich zu Cloud-only-Lösungen wird heute oftmals ein „On Premise“ Ansatz gelebt, bei dem die Hardware für Test- und Entwicklung von und im eigenen Unternehmen gehostet wird. Der große Vorteil: Die Organisation hat alles selbst in der Hand.

Dies ist aber auch zugleich der größte Nachteil: Es entsteht viel Aufwand für Wartung und Betrieb. Mag diese Lösung also für kleine oder mittelgroße Organisationen bzw. Projekte ein gangbarer Weg sein, stößt man bei größeren Unterfangen recht schnell an Grenzen.

Eine andere Möglichkeit sind externe Rechenzentren oder Neudeutsch „private Cloud“, bei denen ein Drittanbieter die Betreuung der Infrastruktur übernimmt. Der Unterschied zu „echten“ Cloud-Lösungen ist, dass in diesem Fall normalerweise nicht nur der Betrieb, sondern auch die Steuerung und der Aufbau vom Drittanbieter übernommen wird. Es muss also für jeden neuen Server oder Cluster ein Antrag beim Drittanbieter gestellt werden, dessen Bearbeitung je nach Verfügbarkeit einige Zeit beanspruchen kann.

Bei der Nutzung von Cloud-Lösungen liegt aber genau hier der Unterschied: Über die einfach zu bedienenden und übersichtlichen Web-Interfaces von Amazon und Microsoft, wird die Steuerung und der Aufbau von der Wartung und Pflege der Hardware entkoppelt.

Die Vorteile liegen auf der Hand – der Anbieter garantiert mehr oder weniger eine 24/7-Verfügbarkeit und Wartung, während der Nutzer es selbst in der Hand hat, wie die Hardware genutzt und kombiniert wird. Natürlich sind auch Hybrid-Ansätze möglich, bei denen in-house gehostete Testumgebungen um zusätzliche Instanzen aus der Cloud erweitert werden. Die Möglichkeiten sind hier sehr vielfältig und benötigen in der Planungsphase besondere Aufmerksamkeit, damit für die Umsetzung in der Cloud der richtige Ansatz gewählt wird und die Testumgebung am Ende dem gelebten Alltag entspricht.

So wird es gemacht: Ein Beispielprozess

Ein beispielhafter Ablauf zur Erstellung von Testumgebungen mittels AWS und Docker bzw. Puppet, kann im Groben etwa so aussehen (siehe Abb. 1):

Zur Bedienung der AWS-Services können, neben der Weboberfläche, auch Commandline Befehle verwendet werden. Somit kann auch das AWS Setup per Script automatisiert werden. Obwohl die von Amazon gewartete AWS Dokumentation ausgesprochen gut ausfällt, ist für Nicht-Techniker die Weboberfläche das geeignete Mittel.

Konfiguration

Im ersten Schritt werden die gewünschten Hardware- und Systemparameter festgelegt (z.B. CPU, RAM, Betriebssystem, IP-Adresse), mit denen eine bestimmte Instanz (sprich Server) ausgestattet sein soll.

Initiale Instanzierung

Mittels dem „run-instances“ Befehl entsteht aus der Konfiguration der neue Server. Auf diesen verbindet man sich per Remote-Desktop Connection, um das Setup durchzuführen.

Setup

In diesem Schritt können nun Docker und Puppet aufgespielt werden, damit dieser Server in weiterer Folge als Zielmaschine dieser beiden Tools verwendet werden kann. Eine AWS-Maschine kann aber auch für anderes zum Einsatz kommen, z.B. als:

- Lizenzserver
- Repository
- Server für ein Drittsystem, welches in der Testumgebung benötigt wird
- Monitoring Instanz

Templates und Template basiertes Rollout

Aus einer AWS-Maschine kann ein Template erstellt werden – es gibt aber auch von der Community vorgefertigte Templates im Marketplace.

Templates können in weiterer Folge wieder mit unterschiedlicher Hardware-Konfiguration als neue Instanz gestartet werden kann. So kann z. B. das Systemverhalten bei Last- und Performancetests auf verschieden dimensionierter Hardware sehr einfach getestet werden. Das Starten der Instanzen - basierend auf den Templates - passiert mit dem Rollout.

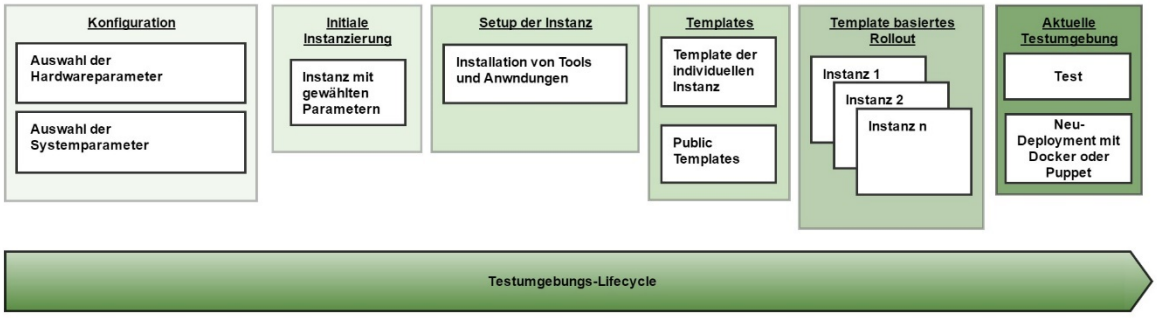


Abb. 1.: Beispielhafter Ablauf zum Setup einer Testumgebung in AWS

Aktuelle Testumgebung

Am Ende des Prozesses hat man nun eine oder mehrere aktuelle Testumgebungen. Ändern sich nun einzelne Systeme, die auf diesen Umgebungen laufen, steht einem flexiblen Neu-Deployment mittels Puppet oder Docker nichts im Weg. Alternativ dazu, muss das zugrunde liegende AWS-Template verändert und der Rollout neu gestartet werden. Bei dieser Variante ergibt sich allerdings der Nachteil, dass der Rollout von vielen Instanzen länger dauern kann, als das Verändern einzelner Systeme mit Puppet oder Docker. Zusätzlich wird bei ständigem neu erzeugen von Instanzen das IP-Adressen Management schnell unübersichtlich.

Sicherheit und Service-Level Agreements

Spricht man in IT-Organisationen die Cloud und deren Verwendung an, werden oft Bedenken bezüglich der Sicherheit geäußert - besonders in Bezug auf Daten. Hier gilt es, sich zu allererst als Organisation bewusst zu machen, welche Daten wirklich sicherheitskritisch sind. Nur so lässt sich ein Cloud-Konzept entwickeln und aufbauen, welches den eigenen Datenschutzanforderungen entspricht.

Kann ein Cloud-Provider möglicherweise mehr Know-How und Ressourcen für Sicherheitsmaßnahmen (z.B. Abwehr von Denial-of-Service- oder Man-in-the-Middle-Attacken) aufbieten als die eigene Organisation? Oft können mit dieser Frage Bedenken, welche die Sicherheit betreffen, zumindest hinterfragt werden.

Eine genaue Studie der Nutzungsbedingungen und SLAs, die man mit der Verwendung von Cloud-Services akzeptiert, ist unverzichtbar. Punkte auf die man dabei achten sollte sind z.B.

- Verfügbare Hosting-Regionen (z.B. Nordamerika, EU-West, Asien)
- Welches Rechtssystem kommt im Problemfall zur Anwendung?
- Wie hoch sind die garantierten Betriebszeiten?
- Wie lange sind geplante Ausfallzeiten?
- Wie schnell wird auf Supportanfragen reagiert?

SLAs sind aber nicht nur gegenüber dem Cloud-Provider von Bedeutung, sondern können auch innerhalb der Organisation eine Rolle spielen. Besonders in großen Unternehmen kann es vorteilhaft sein, als Verantwortlicher für den Betrieb von Testumgebungen, SLAs mit seinen internen Konsumenten (z.B. Projekte) zu vereinbaren.

Organisatorischer Wandel – der Weg zu modernem Testumgebungsmanagement

Von einem organisatorischen Standpunkt betrachtet, verschwimmen bei den bisher vorgestellten Ansätzen und Möglichkeiten die Grenzen zwischen Entwicklung, Test und Betrieb extrem stark. Für die Konfiguration und das zur Verfügung stellen der IaC Komponenten, das Wissen über die Systeme und die Beherrschbarkeit der Abhängigkeiten im Test, werden Kompetenzen aus verschiedenen Bereichen benötigt.

Als Schlagwort für das Zusammenspiel dieser Bereiche hat sich im IT-Jargon in der Zwischenzeit der Begriff „DevOps“ etabliert. Damit sich der DevOps-Ansatz auch tatsächlich in IT-Organisationen etablieren kann, sind einige Maßnahmen unabdingbar:

- Aufbrechen von Team- und Abteilungsgrenzen
- Austausch von Wissen und Information zwischen Development (Entwicklung), Test und Operations (Betrieb)
- Gemeinsame Workshops zur Anforderungserhebung, Umsetzung und Betrieb der Testumgebungen
- Commitment des Management einschließlich Bereitstellung von genügend Raum und Zeit zur Umsetzung
- Aufgeschlossene und technologiebegeisterte Projektmitarbeiter

Natürlich erhebt diese Liste keinen Anspruch auf Vollständigkeit. Im Gegenteil: Sie ist nur die Spitze des organisatorischen Eisbergs, der noch dazu in jedem Unternehmen seine eigene Form hat. Wie bei der Planung der technischen Umsetzung, ist eine umfassende Analyse der Organisation ein essentieller Punkt. Nur so gelingt die Ausarbeitung und Umsetzung der Schritte für den Wandel, um modernes Testumgebungsmanagement erfolgreich zu betreiben.

Autoren

Maximilian Wallisch

Maximilian Wallisch arbeitet seit 2014 bei ANECON Software Design und Beratung G.m.b.H. Er ist dort in den Bereichen Software-Qualitätssicherung und technischer Test tätig. Seine Ziel ist es, Cloud-Computing als Lösung vieler Herausforderungen im Softwaretest-Umfeld zu etablieren.

maximilian.wallisch@anecon.com

Dietmar Berchtold

Dietmar Berchtold arbeitet seit 2009 bei ANECON Software Design und Beratung G.m.b.H. Er ist dort im Bereich Test Center & Test Organisation tätig. Er hat in vielen unterschiedlichen Projekten Erfahrung mit der Organisation und dem Betrieb von Testumgebungen machen können. Das Entdecken neuer Tools und deren Möglichkeiten ist seine große Leidenschaft.

dietmar.berchtold@anecon.com

Mobile Apps Performance Testing

Using Open Source Tool JMeter

In the past, mobile played a minor role in business applications but nowadays with the evolution of smartphones and tablets, mobile applications and mobile websites have become a major channel for conducting business, improving employee efficiency, communicating, and reaching consumers. Each and every business aspect including banking, ecommerce, retail sector, social networking is facing the digital transformation and customer wants everything within grab of their hands and also within fraction of seconds.

With the changing scenario and shifting of user base from conventional desktop to mobile interface, the requirement of performance testing for mobile applications not just a nice thing to have, but an essential activity for success in the age of the instant-on services. The demand for better performance generates continuous requirements for enhanced testing approach to deal with the complexities challenges of mobile performance testing in low cost.

Today, performance problems with mobile applications lead directly to revenue loss, brand damage, and diminished employee productivity. It's now mandatory for businesses to ensure the performance of applications in their mobile environments.

We have developed mobile performance testing solution based on open source tool JMeter which will reduce the testing, improve overall performance of the application by testing the performance of applications and mobile websites simulating real world conditions and boost the confidence in application for go live. The ease of implementation and flexibility of the solutions helps to customize it according to your needs. All of that have an effect on quality. Test teams are responsible for maintaining good quality and distributed agile teams bring up the challenge whether we are able to persist under pressure. Test teams are in most of the cases distributed. It is very common to have test engineers as contractors from other side of the world. How does this affect overall progress? After all, the adage says, “Out of sight out of mind”, but the quality cannot be ignored. Hey, we are separated but for sure not divorced!

What are Mobile Apps?

Mobile apps are the programs designed to run on Mobile devices.

Purpose of Mobile Apps

There can be various purposes of a Mobile Application and below are some common listed purposes:

- Information Retrieval
- Business
- Communication
- Education
- Finance
- Social Networking
- Mobile Commerce
- Health and Fitness
- Travel and Transport

And many more…

Types of Mobile Applications

There are majorly three types of Applications:

Web Based Application

Features:

- Stored on a remote server
- Delivered over the internet through a browser
- Websites that look and feel like native applications.

Native Applications

Features:

- Developed for use on a particular platform or device
- Specific programming language, such as Objective C (for iOS) and Java (for Android)
- Provides fast performance and a high degree of reliability

Hybrid

Features:

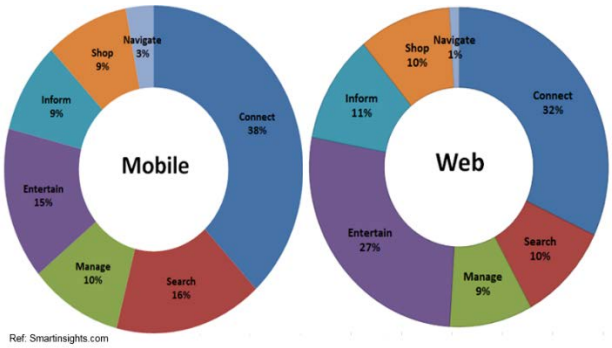
- Hybrid Apps are like native apps, run on the device, and are written with web technologies (HTML5, CSS and JavaScript)

- Leverages device's browser engine but not browser, a web-to-native abstraction layer
- Access to device capabilities that are not accessible in Mobile Web applications, such as the accelerometer, camera and local storage.

Importance of Mobile Performance Testing

Current Trends

Mobile is dominant in major areas.



Mind your business; through mobile

Business

What ecommerce did to retail, mobile is doing to ecommerce.

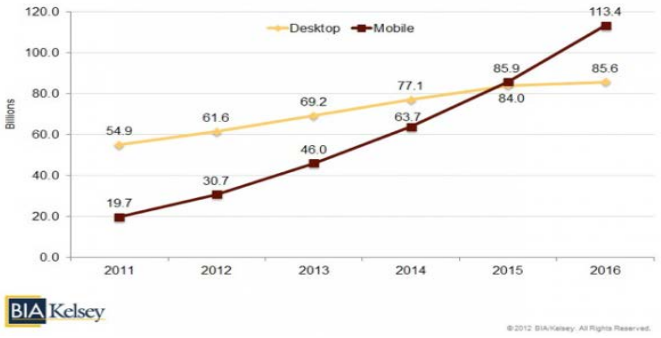
Ecommerce is changing to mobile commerce

User Behaviour

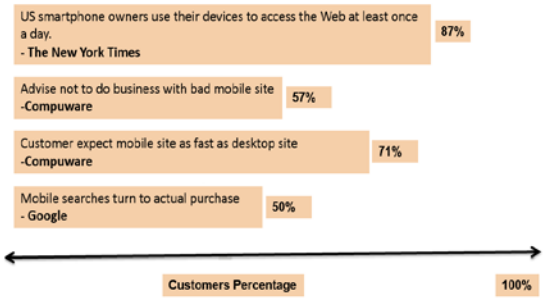
Analyst firm BIA/Kelsey has projected

By 2015 there will be more local searches coming from smartphones than PCs in the US.

Smartphone search volumes are growing faster than search on the PC.



What researchers found about User Behaviour?



Challenges in Mobile Performance Testing

Approach 1: Using Real Devices to Generate Load

Challenges:

- Testing Cost will increase drastically
- Manage device collection is hectic
- Poor Network Quality, High Latency

Approach 2: Using Licensed Tools

Challenges:

- Tools use emulators to generate script
- Tools supports only browser based recording
- Restriction on number of users
- License Cost

Approach 3: Using Network Proxy Capturing Utilities

Challenges:

- Requires lot of effort in filtering out the requests and responses from Application under test
- Script generation and enhancement is complex

- Script recorded not close to real world
- Supports Licensed tools

Using JMeter for Mobile Performance Testing

Fact

All the performance testing tools work on one abstract idea.

"Capture the HTTP traffic coming and going from mobile device."

We have to simulate the same using JMeter but alas no sampler request for mobile protocols so this cannot be captured in the conventional way.

Concept

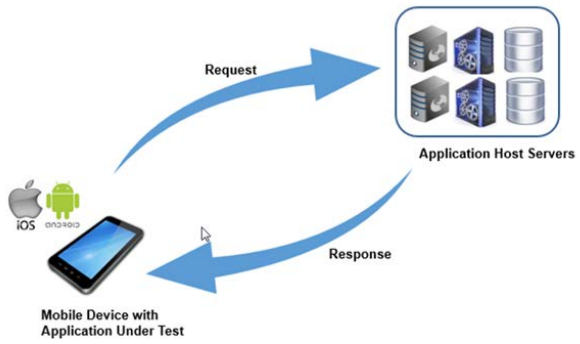
Native and hybrid mobile application uses the HTTP protocol to communicate with the server and this can be utilized.

Approach

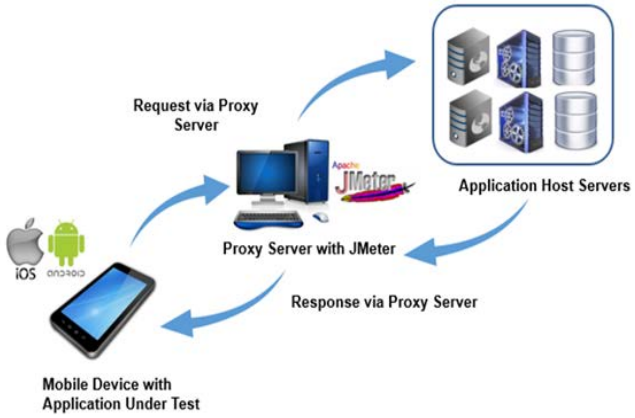
- Make the machine with JMeter a proxy server
- Connect mobile device with this proxy server
- Capture the HTTP traffic (request – response from mobile device) using proxy server as all the traffic from device is now routed to this proxy server

This approach is far closer to the real world traffic simulation as compare to that of emulators.

The below diagram shows exchange of requests and responses between Mobile app on device and Mobile Application server.



The below diagram shows exchange of requests and responses between Mobile app on device and Mobile Application server via proxy server with JMeter to capture traffic.



Script Generation Steps for JMeter

1. Select a New Test Plan > Thread Group.
2. Add a Non-Test element > HHTP Proxy Server and change the port to 8888.
3. In Internet Explorer, select Tools > Internet Options > Connections. Click LAN settings and enter the port and address of the device Internet proxy.
4. Select Start Server on JMeter.
5. On the device, delete browser cache data which includes Temporary Internet Files and Cookies.
6. It will start capturing the HTTP requests and response.

We have successfully used JMeter to load test Mobile Native Applications based on Android and iOS.

- **Application 1:** This is an Android platform based native application and enables individuals and families to safely maintain their medical records and other important documents in one central location and instantly access them anytime from anywhere in the world using the Internet.

- **Application 2:** This is an iOS based Native Application and serves as a mobile wallet to end users and he can add merchants, billers and pay them using this application.

We have used this approach across multiple projects of different sizes. The approach is independent of project size or team size.

Benefits

The major benefits of this approach are listed below.

- Device independent
- Application platform independent
- Applicable to other projects situation without any limitations
- Scripts generated is much closer to real world as compared to that of using simulators
- No network latency or overhead is induced by using network traffic capture tools

Assumptions

The following assumptions are valid for this approach.

- The native application is interaction with the server via HTTP protocol.
- Device with the application installed is available.
- The user's actions that do not have ant HTTP request sent and received will not be captured and as these are executed at the client level are not of as much importance to the performance testing.

Dependencies

The following technical dependencies have been identified.

- The device and machine with tools installed should be in same network.
- We have sufficient privileges to make the machine with tool installed a proxy server.

References

http://jmeter.apache.org/

http://jmeter.apache.org/usermanual/

http://www.smartinsights.com/

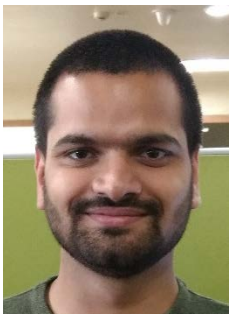
https://www.wikipedia.org/

- 'The Every Computer Performance Book' by Bob Wescott
- 'Applications Performance Symptoms and Bottlenecks Identification' from www.agileload.com

Appendix

- Apps: Application
- AUT: Application Under Test
- Perf: Performance

Author



Devendra Singh

Performance Test Lead working for Nihilent Technologies Ltd.

Devendra has more than 5.5 years of experience in Performance Testing. He is working as a key resource in Testing CoE of Nihilent Technologies Limited and his role is to look for innovative solutions and implement that in projects to increase efficiency and reduce costs. Earlier in his career, he has worked with IT giants like Capgemini and Fiserv with same kind of role and got Bronze Star award and Value in Performance award for his innovative solutions.

He is passionate about learning new tools and technologies coming in market and developed various accelerators to assist and reduce turnaround time in Performance testing. He holds various certifications like HP Loadrunner AIS Certification, Neotys Certified Professional and ISTQB Certified Tester.

Devendra is also a blogger and actively participates in online forums to learn and assist others.

Email: devendra.singh@nihilent.com
devendrasinghhbti@gmail.com

Mobile: +91-9595971317 / +91-7350671409

Blog: http://performancetestingcentre.blogspot.in/

Twitter: @DevendraSingh08

LinkedIn: https://in.linkedin.com/in/devendrasinghlr

Systemisches Projektmanagement

Anwendung systemischer Regeln in der Projektarbeit

Dieser kurze Artikel dient als kleiner Appetit-Happen auf die systemische Denkweise und Haltung, zu denen ich Ihnen in meinem Workshop am 20.01.17 einen ersten Zugang verschaffen werde. Führungskompetenz ist eine Dauerbaustelle, zumindest bei mir. In meinem Bestreben, den einen oder anderen Missstand zu beheben, beschäftige ich mich seit vielen Jahren mit der menschlichen Seite des Projekterfolgs und stieß dabei auf eine Goldgrube der Erkenntnis: die Systemische Denkweise.

Was verbirgt sich hinter diesem Begriff? Bei der Beantwortung dieser Frage halte ich es wie der alte Pauker in dem Uraltfilm „Die Feuerzangenbowle“: Da stellen wir uns mal janz dumm. Wat is na eine systemische Denkweise? (Wer die Szene nicht kennt, siehe Google: Feuerzangenbowle Dampfmaschine.)

Am besten wir nehmen dazu ein Projekt als Beispiel. Ein Projekt ist ein System, weil es sich gegenüber seiner Umgebung u.a. durch ein Ziel und Personen abgrenzt, die mehr oder weniger freiwillig dem Projekt zugeteilt wurden. Ein wichtiger Aspekt der systemischen Denkweise ist es nun, dass man davon ausgeht, dass die Projektmitarbeiter sich ständig gegenseitig in ihrem Verhalten beeinflussen. Beispiel: Es gibt einen Kollegen, der sich am liebsten einigelt, um ungestört über eine Lösung zu grübeln. Ein anderer wiederum ist bestrebt, seine Aufgaben durch intensive Kontaktaufnahme mit seinen Leidensgenossen zu lösen. Bringen Sie die beiden Typen zusammen, wirkt das möglicherweise wie Nitro und Glyzerin. Bumm! Wer ist schuld?

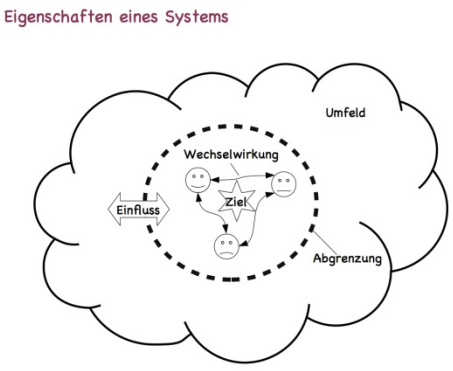


Bild: In Systemen wirken vielerlei Einflüsse und Wechselwirkungen, die in ihrer Gänze weder verstehbar noch erkennbar sind.

Jetzt bringe ich noch einen weiteren systemischen Gesichtspunkt ins Spiel: Jedes System ist in ein Umfeld eingebettet, das auf dieses System wirkt und umgekehrt. Stellen Sie sich also weiter vor, dass dieses Projekt in einem Unternehmen durchgeführt wird, das großen Wert auf Teamarbeit und Kommunikation legt. Der Projektleiter ist geradezu das Idealbild dieser Teamkultur. Auf welche Seite wird er sich wohl schlagen, wenn der oben angedeutete Konflikt den Projektfrieden gefährdet? Wer fühlt sich im Recht? Wie reagiert wohl unser kontaktscheuer Einsiedler, wenn die anderen ihn zu mehr Kommunikation bekehren wollen? Wahrscheinlich zieht er sich noch mehr in sein Schneckenhaus zurück oder verlässt genervt das Projekt. Dummerweise hat aber gerade diese Person den Schlüssel zum Projekterfolg im Kopf. Das Projekt scheitert ohne ihn. Wer ist schuld?

Sie sehen, dass uns die Schuldperspektive nicht wirklich weiter bringt. Trotzdem versucht unser Verstand oft krampfhaft, die Welt auf der Suche nach Schuldigen bis zur Unkenntlichkeit zu vereinfachen. Wir machen dabei nicht selten aus einer Mücke einen Elefanten. Der wirft dann so einen riesigen Schatten, dass wir die darin verborgenen Zusammenhänge nicht mehr erkennen.

Bescheidenheit als Erfolgsfaktor

Die systemische Denkweise wirkt dieser Wahrheitsverzerrung entgegen. Jetzt wird es ein bisschen kompliziert: Die Denkweise geht davon aus, dass wir die Wechselwirkungen in Projekten und Unternehmen nicht wirklich verstehen können. Sie sind einfach zu kompliziert und vielschichtig, weil Menschen nun mal sehr kompliziert und vielschichtig sind – vor allem, wenn sie mit anderen in Beziehung stehen. Wer schon nicht mal seine eigenen Kinder und Ehepartner versteht, braucht sich nicht einzubilden, seine Kollegen in jeder Situation zu durchschauen. Mal ehrlich, inwieweit durchschauen Sie sich selbst? Also vergessen Sie es! Wir können Projekte nicht wirklich verstehen! Was nun?

Eine gewisse Bescheidenheit ist geboten. Was wir tatsächlich tun können ist, dass wir das Projekt bzw. die Menschen darin ein bisschen „anstupsen“ und beobachten, was passiert. Im Coachjargon heißt das: das System anregen oder stören. Bewegt es sich in der gewünschten Richtung, scheint der Stupser funktioniert zu haben. Aber Vorsicht, das heißt nicht, dass Sie jetzt voll ausholen können, um das System ein für allemal auf die Spur zu bringen. Hier gilt der Grundsatz von Paracelsius, es kommt auf die Dosis an. Es heißt auch nicht, dass Sie jetzt ein sicheres

Erfolgsrezept für Projekte gefunden haben. Es heißt nur, dass es unter den gegebenen Umständen, die wir ja nicht wirklich kennen, momentan funktioniert, und dass es vielleicht in ähnlichen Situationen unter Umständen wieder klappen könnte. Sie lesen buchstäblich die Unsicherheit im vorangegangenen Satz. Wenn wir allerdings das Projekt nach systemischen Prinzipien beobachten und anregen, dann erhöhen wir die Wahrscheinlichkeit, dass wir positive Impulse geben. Das klingt jetzt nicht besonders hilfreich für alle, die auf den Stein der Weisen gehofft haben. Hier kann ich gleich ein weiteres systemisches Prinzip ins Spiel bringen: Anerkennung des Gegebenen. Und gegeben ist: Es gibt keine Garantien. Wir wissen in Projekten nie, was wirklich passiert.

Ich hoffe, Sie sind jetzt neugierig darauf, wie systemischen Regeln Ihnen dabei helfen können, Ihre Erfolgswahrscheinlichkeit zu erhöhen. Sie werden erstaunt sein, wie einfach diese Regeln und die Prinzipien dahinter sind. Doch auch hier ist Bescheidenheit angebracht. Nur weil Regeln und Prinzipien einfach zu verstehen sind, sind sie noch lange nicht einfach zu leben. Denn dazu müssen wir sie verinnerlichen. Das erfordert viel Übung und Geduld sowie die Einsicht, dass es weder Erfolgsgarantien noch Perfektion gibt. Der systemische Ansatz beruht auf dem beharrlichen und geduldigen Bestreben, Lücken zu schließen in dem Wissen, dass dies nie völlig und dauerhaft gelingen kann. Wenn Sie sich dieser Wahrheit mutig stellen, haben Sie schon mal eine wichtige Voraussetzung für gute Führungsarbeit erfüllt.

Systemische Prinzipien

Hier drei Beispiele, wie wir wichtige systemische Prinzipien auf Projekte anwenden können:

Anerkennung des Gegebenen: Schauen Sie den Tatsachen ins Auge. Sorgen Sie u.a. dafür, dass möglichst viele relevante Aspekte des Projekts für Sie und das Team transparent sind. Jeder sollte beispielsweise Ziel, Weg und beteiligte Personen und Rollen, aber auch relevante Unsicherheiten, Risiken, Konflikte und Defizite kennen. Projektmanagement funktioniert nur, wenn wir in der Realität agieren. Die Kunst des Projektmanagements beruht nicht darauf, die Realität im Sinne eines Wunschbilds hinzubiegen, sondern aus der Realität das Beste zu machen. Hört sich einfach an. Wer schon länger in der Projektmühle steckt, weiß, wie schwer es ist, dieses Prinzip konsequent zu leben. Was die Dinge erschwert ist die Tatsache, dass wir unsere Wahrnehmung eines Projektes nur der Realität annähern können. Es bleibt immer eine Lücke.

Recht auf Zugehörigkeit: Jedes Projektteam benötigt Vereinbarungen, Regeln und Rituale, die die Zugehörigkeit der Teammitglieder zum Projekt sichtbar machen. Eine wichtige Maßnahme ist es dabei, sicherzustellen, dass jede Person Ihren Platz (Rolle, Funktion) im Team kennt und so in die Kommunikation eingebunden wird, dass sie eine faire Chance hat, ihren Beitrag zum Teamerfolg zu leisten. Ein weiterer wichtiger Grundsatz in diesem Zusammenhang ist, dass wir abwesende Teamangehörige so in unser Handeln einbeziehen, als wären sie anwesend (Stichwort: Meckern hinter dem Rücken der Betroffenen). Auch hier gilt, dass wir uns in unserer Menschlichkeit einem Idealzustand nur nähern können und immer wieder so unsere „Aussetzer“ haben. Wer hier eine weiße Weste hat, bitte melden.

Achtung informeller Hierarchien: Neben der formalen Hierarchie des Organisationsplans existieren weitere informelle Hierarchien, die beachtet werden sollten, um Konflikte zu vermeiden oder zu lösen. Beispielsweise ist es wichtig, dass Projektleiter, die jünger sind als ihre Teammitglieder, verstehen, dass sie diese Tatsache in angemessener Weise würdigen. Damit steigt die Wahrscheinlichkeit, dass die Älteren wiederum die Führungsposition des Jüngeren achten. Auch hier ist es angebracht, die Kirche im Dorf zu lassen. Der Jüngere darf ruhig mal provokant sein und der Ältere besserwisserisch, solange die Regeln der Höflichkeit und Fairness gewahrt bleiben oder wiederhergestellt werden.

Mich selbst beeindruckt die Kombination von Weisheit und Bescheidenheit, die dieser systemische Ansatz bietet. Ich denke, dass diese Kombination ein Schlüssel für erfolgreiche Führungsarbeit und Mitarbeiterzufriedenheit ist. Ich hoffe in aller Bescheidenheit (die mir leider nicht immer in dem Maße gegeben ist, wie ich es mir wünsche), ich konnte Sie ein bisschen neugierig auf Systemisches machen. Lernen Sie mit der Lücke zu leben, indem Sie sie als Raum für Ihre nie endende persönliche Weiterentwicklung und Bereicherung annehmen.

Wenn Sie noch mehr über systemische Regeln und Prinzipien erfahren wollen, senden Sie mir eine E-Mail mit dem Stichwort „systemisch“ an info@die-menschliche-Seite.de. Ich freue mich auch über Ihre Anregungen und Rückmeldungen zu diesem Thema.

Autor

Dipl.-Ing. Peter Siwon

Dipl.-Ing. Peter Siwon kennt die Projektarbeit aus vielen Perspektiven: Forschung, Entwicklung, Projektleitung, Schulung und Beratung, Vertrieb, Marketing und Geschäftsführung. Er ist Mitgründer des Embedded Software Engineering Kongress sowie Kolumnen- und Buchautor. Er lehrt an der University of Applied Science in Regensburg, Nordhausen und Braunschweig. Mehr über ihn finden Sie unter www.systemisches-projektmanagement.info

Vom Nutzer her denken

Mit Design Thinking erfolgreich aus dem Verdrängungswettbewerb aussteigen und Nutzerbedürfnisse gezielt erfüllen

Der Ruf nach Innovation und kreativen Ideen in Unternehmen wird immer lauter: Die Angst vor dem Verdrängungswettbewerb geht um, denn mit Me-too-Produkten und -Dienstleistungen ist ein langfristiges Bestehen nicht mehr möglich. In diesem Kontext sind der Nutzer und seine Entscheidungen die wichtigsten Faktoren innerhalb und außerhalb des Unternehmens – aber sie werden durch die Fokussierung auf den Return on Investment und das Erreichen der Geschäftsziele nach wie vor viel zu oft vergessen.

Eine echte Alternative dazu, statt ständig auf die Konkurrenz oder die neueste Marktentwicklungsergebnisse zu schielen, ist es, den tatsächlichen Bedarf vom Nutzen des Nutzers bzw. Kunden her zu analysieren und zu bedenken. Dafür eignet sich wohl kaum eine andere Methode so sehr, wie die des Design Thinkings. Design Thinking bindet Mitarbeiter aus unterschiedlichen Bereichen und auch den Kunden selbst ein, die dann – möglichst ganzheitlich denkend – entsprechend kundenorientierte Lösungen entwickeln sollen. Die notwendigen Zutaten für einen solchen Denkprozess: Empathie für den Nutzer und das unterschiedliche Erfahrungswissen der Mitarbeiter aus den verschiedenen Wirkungsbereichen.

Der Prozess des Design Thinking ist zunächst neu und gewöhnungsbedürftig, weil er nicht linear verläuft: In einer Art „Denkspiralen“ nähert sich das Team der bestmöglichen Lösung und generiert dabei Wissen, um dann am Ende ein bereits getestetes, optimiertes und Erfolg versprechendes Produkt oder eine entsprechende Dienstleistung zu präsentieren.

Was genau bedeutet Design Thinking?

Design Thinking hat nur am Rande mit dem zu tun, was landläufig unter dem Begriff „Design“ verstanden wird. Es geht hier weniger um das Äußere eines Produktes, sondern vielmehr um die Funktion und Wirkung von Dingen und Prozessen. Das kann genauso ein Verkaufsgespräch betreffen wie auch die Eigenschaften eines neuen Produkts. Der offene Design Thinking-Prozess hat aber immer das Ziel, eine kundenorientierte Lösung zu finden - das konkrete Ergebnis ist dabei zu Beginn unbekannt. In einer lockeren Abfolge von verschiedenen Phasen, in denen unterschiedliche Techniken eine Rolle spielen, nähert das Team sich der optimalen Lösung. Damit das funktioniert, ist eine bestimmte Kombination von Denkweisen und Methoden notwendig:

Wichtig: Das Mindset

Nur die Kombination aus der richtigen Einstellung und der Abfolge bestimmter Schritten entfesselt das Denken einer Person. Innerlich sollte das Denken aller Beteiligten auf Wachstum ausgerichtet sein, statt nur auf das Halten eines mäßig erfolgreichen Status Quo. Damit das gelingt, sind vier Faktoren von Bedeutung:

Empathie: Der Wunsch, Dinge nachhaltig zu ändern, kommt aus dem echten Bedürfnis, seine Kunden zu unterstützen. Es geht darum, ein Verständnis aufzubauen und verschiedene Gefühle und Bedürfnisse wahrzunehmen, emphatisch zuzuhören und dann die unterschiedlichen Perspektiven in den Prozess zu integrieren.

Motivation: Innere Motivation ist der Brennstoff für Kreativität. Sie müssen ein echtes Interesse, Begeisterung und Engagement für Ihre Arbeit spüren. Ob es die Herausforderung ist, ein komplexes Problem zu lösen, eine brennende Frage zu beantworten oder ob Sie etwas entwickeln wollen – es geht immer darum, sich aktiv zu engagieren.

Offenheit: Offenheit für neue Ideen, Perspektiven und Wege. Dazu gehören auch eine aktive Vorstellungskraft, Sensibilität und Aufmerksamkeit, die Vorliebe für Vielfalt, intellektuelle Neugier und die Fähigkeit, selbst Dinge zu beurteilen. Denn ohne ein offenes Wesen wird es schwierig, das Potenzial einer Veränderung, also „die Luft nach oben“, in vollem Umfang zu realisieren.

Optimismus: Eine optimistische Einstellung ist für den Umgang mit kritischen Stimmen im Prozess selbst sehr hilfreich. Vor allem, wenn es darum geht, nach vorne zu blicken und kreative und produktive Lösungen zu finden. Der *Design Thinking*-Prozess ist sehr dynamisch und kann viele unerwartete Wendungen nehmen, bevor eine bahnbrechende Lösung erreicht wird; Optimismus feuert dann Flexibilität und Ausdauer an.

Unverzichtbar: Die Rahmenbedingungen

Für Design Thinking gibt es keine Formel – aber ein paar wichtige Rahmenbedingungen:

Verständnis und Bedürfnisfindung im Fokus

Um für Kunden einen Mehrwert zu schaffen, müssen Sie in erster Linie deren Motivation auf einer tieferen Ebene verstehen. Erst dadurch können unerfüllte Bedürfnisse entdeckt und neue Möglichkeiten entwickelt werden.

Interdisziplinäre Zusammenarbeit

Der Wert von Design Thinking basiert auf den verschiedenen Perspektiven und Eigenschaften der einzelnen Teammitglieder. Das fördert die kreative Energie und etabliert eine Eigenverantwortung. Dabei ist es wichtig, dass in der Zusammenarbeit offen, einfühlend und neugierig gehandelt und gedacht wird.

Systemisches Denken

Design Thinking sucht die Lösung in einem System, das aus verschiedenen Teilen besteht: aus der Perspektive der Stakeholder, der Unternehmensstrategie und der Vision. Wenn Sie diese Verbindungen genauer betrachten, wird Ihnen auffallen, dass das Unternehmen einer Landkarte ähnelt, die voller Verbindungen und Abhängigkeiten ist. Wenn Sie die Komplexität herausnehmen und sich auf den Kundenutzen fokussieren, erkennen Sie schnell die Ansatzpunkte, die den meisten Wert schaffen können.

Prototyping und Experimentieren

Ein Prototyp hilft dabei, Ideen greifbar zu machen. Gleichzeitig dient er als Werkzeug für das Denken, die Kommunikation mit anderen und die Weiterentwicklung.

Praxischeck: So kann Design Thinking funktionieren

Wichtig für die Praxis ist zu beachten, dass alle Prozessstufen nicht linear sind, sondern gleichzeitig auftreten und wiederholt werden können.

1. Einfühlen

Um etwas für einen Kunden zu entwerfen, müssen Sie ein Gespür dafür entwickeln, wer er genau ist und wie er oder sie die Welt sieht. Am besten schaffen Sie das durch Beobachtung. Dabei können Sie erkennen, was er wirklich tut, und lassen sich nicht von dem, was er sagt, das er tut, blenden. Das Endprodukt einer solchen Beobachtung ist oft eine starke Kombination aus visuellen Elementen wie Fotos oder Zeichnungen und aus schriftlichen Erzählungen, die alle Beobachtungen aufgreifen und so eine erste Perspektiven eröffnet, die in Richtung einer kundenfreundlichen Lösung weist.

2. Definieren

In diesem Schritt bringen Sie das eigentliche Problem, das Sie anhand Ihrer Bemühungen lösen wollen, explizit zum Ausdruck. Um wirklich überzeugende Ideen liefern zu können, brauchen Sie nämlich zuerst die richtige Fragestellung. Das bedeutet: „Entwerfen Sie einen Bürostuhl“ funktioniert nicht; „Finden Sie einen Weg, um den Kunden in seinem langen Büroalltag bestmöglich zu unterstützen“ aber schon. Das Ziel dieser Phase ist, das wirkliche Problem zu finden und es dann so zu formulieren, dass es kreative Lösungen herausfordert.

3. Ideengenerieren

Wir neigen dazu, Ideen zu vertrauen, von denen wir wissen, dass sie funktionieren, z. B., weil wir das schon mal erlebt haben. Dieses natürliche Verlangen nach Sicherheit können Sie auch anders bedienen: Entwickeln Sie ganz viele unterschiedliche Ideen, um sicher zu sein, dass mindestens eine funktionieren wird. Erstellen Sie einen ganzen Katalog und seien Sie nicht auf eine oder mehrere fixiert. Schalten Sie Ihre unbewussten Filter aus bzw. machen Sie sich diese Filter bewusst: Sie müssen spüren, wann Sie und Ihr Team Ideen generieren – und wann Sie diese Ideen bewerten und aussortieren. Dabei sind die verschiedenen Perspektiven der verschiedenen Personen zusätzliche Schlüsselelemente: Sie bekommen wesentlich bessere Antworten, wenn fünf Personen einen Tag lang ein Problem bearbeiten als wenn eine Person fünf Tage an einem Problem sitzt.

4. Prototyping und testen

In diesem Schritt generieren Sie aufgrund einer skizzierten Lösung einen Prototyp. Dieser kann durch alles mögliche dargestellt werden: Ene Wand voller Post-its, ein Rollenspiel, ein Objekt, ein User-Interface oder ein Storyboard – alles ist möglich. Prototypen funktionieren dann am besten, wenn alle Beteiligten sie hautnah erleben und mit ihnen interagieren können. Was dabei gelernt und erfahren wird, hilft dabei, noch mehr Empathie zu erfahren und noch erfolgreichere Lösungen zu erarbeiten.

Durch das Testen wird der Prototyp verfeinert. Testen ist eine weitere tolle Möglichkeit, Empathie durch Beobachtung aufzubauen und sich noch mehr einzubringen. Oft ergeben sich in dieser Phase vollkommen neue Einsichten. Tests ermöglichen auch das Finetuning unseres Blickwinkels. Manches Mal zeigen die Tests, dass wir nicht die Lösung erarbeitet haben, sondern dass wir die Problemstellung ändern müssen.

Prototyping ist unverzichtbar, denn ein Prototyp sagt mehr als tausend Bilder und löst Meinungsverschiedenheiten, weil er Missverständnisse auf ein Minimum reduziert. Darüber hinaus ermöglicht er es, schnell und billig zu scheitern: Sie können eine Reihe von Ideen ausprobieren, ohne zu viel Geld und Zeit in eine Überprüfung ste-

cken zu müssen. Außerdem macht er die Entwicklung einer Gesamtlösung handhabbar: Sie teilen das Problem in mehrere kleine Schritte und können immer wieder Testphasen zwischenschalten. Und last but not least bietet er eine gute Möglichkeit, den Kunden damit anzusprechen.

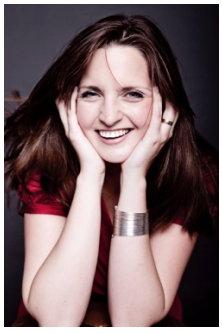
Fazit

Design Thinking als Methode ist nicht schwer umzusetzen und es ist nicht völlig neu – einige Unternehmen nutzen diesen Ansatz bereits. Aber für die meisten Unternehmen erfordert dieser Prozess eine neue Ideologie, weil er sich auf qualitative und frisch gedachte Daten stützt, anstatt reine Statistik als Basis zu nehmen. Der Prozess stellt den Nutzer und deren Bedürfnisse in den Mittelpunkt. Dazu ist die Identifizierung mit und den Glauben an neue Erkenntnisse notwendig, die subjektiv sind und in ihrer Mehrdeutigkeit auch Risiken bergen und Mut erfordern. Aber dafür gewinnen Sie auch neue und frische Erkenntnisse, die außer Ihnen niemand hat, und werden jenseits von Benchmarking, Marktforschung und Gleichmacherei erfolgreich sein.

Bibliographie

GERSTBACH, I. (2016): Design Thinking im Unternehmen. Ein Workbook für die Einführung von Design Thinking. GABAL Verlag: Offenbach

Autorin



Mag. Ingrid Gerstbach

Ingrid Gerstbach ist Expertin für Design Thinking und Innovationsmanagement, Wirtschaftspsychologin und Unternehmensberaterin. Sie sieht sich als Entwicklungshelferin für Unternehmen, um Innovationen, neue Erfolgspotenziale und nachhaltige Wertschöpfung zu ermöglichen.
www.ingridgerstbach.com

Ingrid.gerstbach@designthinking-wien.at

Strukturierte Tests bei defizitärer Dokumentation

Wie man zwei Fliegen mit einer Klappe schlägt

Kennen Sie diese Situation? Unter hohem Zeitdruck soll ein Abnahmetest für ein historisch gewachsenes System vorgenommen werden. Die Ansprüche an die Testabdeckung sind hoch, die Dokumentation des Systems aber stellenweise leider lückenhaft. Dies sind häufig die Vorzeichen, unter denen fachliche Abnahmetests stattfinden.

Unsere Antwort auf diese Ansprüche ist eine spezielle Methodik der Testdokumentation, die eine Prüfvorschrift und eine daraus abgeleitete Testspezifikation vorsieht.

Mit dieser Methodik

- **werden Dokumentationslücken geschlossen,**
- **wird die Festlegung einer angemessenen Testabdeckung unterstützt,**
- **wird eine systematische Ableitung von Testfällen erleichtert und**
- **wird die Generierung von Testdaten vorbereitet.**

Einsatzszenarien

Diese Methodik hat sich für das Testen von Systemen bewährt, die über verschiedene Schnittstellen Daten erhalten, verarbeiten und weiterreichen. Im Folgenden dient das Online-Banking-Portal einer Bank als Beispielanwendung. Jedoch ist die Methodik natürlich auch auf andere Systemtypen übertragbar, z. B.

- das Einkaufsportal eines Händlers,
- das Buchungssystem eines Hotel- oder Reiseanbieters,
- das Online-Portal einer Versicherung,
- ein SAP-System,
- etc.

Mit Tests sind in dem vorliegenden Artikel **fachliche Abnahmetests** gemeint: Die hauseigene Software-Entwicklung oder ein beauftragter Dritter haben ein System entwickelt oder weiterentwickelt, das nun von der Fachabteilung getestet und abgenommen werden soll. Erst nach der Abnahme durch die Fachabteilung darf das System in die Produktion übernommen werden.

Zu diesem Zeitpunkt sind die entwicklungsbegleitenden Tests, d.h. die Tests der Software-Entwicklung, die unter Kenntnis des Quellcodes vorgenommen wurden, bereits abgeschlossen, und das System ist aus Entwicklungssicht fertig.

Mit den fachlichen Abnahmetests soll nun überprüft werden, ob die Vorstellungen der Fachabteilung korrekt umgesetzt wurden. Dabei ist es hilfreich, sich die folgenden drei Fehlerursachen vor Augen zu führen:

1. Während der Umsetzung wurde ein Fehler gemacht.
2. Eine Vorgabe der Fachabteilung wurde falsch verstanden und entsprechend „fehlerhaft“ implementiert.
3. Die Vorgaben der Fachabteilung waren lückenhaft und wurden in der Implementierung möglichst sinnvoll geschlossen.

Schon an diesen drei Beispielen sieht man sehr gut, dass die fachlichen Abnahmetests unverzichtbar sind:

Fehler auf Grund der ersten Fehlerursache können durch eine entwicklungsbegleitende Qualitätssicherung gefunden werden. Hierfür sind fachliche Abnahmetests nicht zwingend notwendig.

Fehler auf Grund der beiden anderen oben genannten Fehlerursachen können dagegen nicht durch eine rein entwicklungsbegleitende Qualitätssicherung gefunden werden.

Der **fachliche Abnahmetest** unterscheidet sich von den **entwicklungsbegleitenden Tests** nicht nur dadurch, dass in dem einen Fall die Software-Entwicklung und in dem anderen Fall die Fachabteilung bzw. externe Testspezialisten testen, sondern auch dadurch, dass bei den fachlichen Abnahmetests nur die äußeren Schnittstellen in den Test einbezogen werden. Es handelt sich bei den fachlichen Abnahmetests also um sogenannte Black-Box-Tests.

Betrachten wir beispielhaft ein System, das Bankleitzahl und Kontonummer in eine IBAN umrechnet und hierfür eine Web-Oberfläche anbietet. Für den fachlichen Abnahmetests steht nur die Web-Oberfläche zur Verfügung. Zum Test des Systems müssen geeignete Kombinationen aus Kontonummer und Bankleitzahl gewählt werden, um die Korrektheit des Systems angemessen zu testen.

Beim entwicklungsbegleitenden Test sind alle internen Bibliotheken und Komponenten bekannt und können separat getestet werden.

In der Regel werden die fachlichen Abnahmetests teilweise manuell und teilweise automatisiert ausgeführt. Mit der Zeit entstehen Regressionstests, die bei jedem neuen

Release immer wieder ausgeführt werden, um zu gewährleisten, dass die Bestandsfunktionen weiterhin korrekt arbeiten.

Im Test werden Daten (**Testdaten**) an das zu testende System übergeben und **Ergebnisdaten** von dem System empfangen und auf Korrektheit geprüft.

Die Testdokumentation

Eine Testdokumentation deckt je nach Größe des Projekts u.a. die folgenden Themen ab:

- Beschreibung der Testfälle einschließlich der Ergebniserwartung
- Aufbau der Testumgebung
- Organisatorische Aspekte wie etwa die zeitliche Planung oder die Ressourcenplanung
- Vorgaben zum Fehlertracking
- Dokumentation der Testdurchführung und der Nachtests im Fehlerfall
- Testabschlussbericht

Unsere Methodik befasst sich mit dem ersten Aspekt.

Die zugehörige Testdokumentation erfolgt hierbei in Form von zwei Dokumenten,

- der **Prüfvorschrift** sowie
- der **Testspezifikation**.

Die Prüfvorschrift

Beginnen wir mit einer Definition:

> Die Prüfvorschrift ist ein Dokument, das
>
> - die fachlichen Anforderungen an das Testobjekt aus Testsicht beschreibt und
> - alle notwendigen Vorgaben für die Testspezifikation enthält.

An die Prüfvorschrift stellen wir die folgenden Anforderungen. Diese Anforderungen werden in den nachfolgenden Abschnitten detailliert erläutert.

- Beschreibung der Architektur und der Schnittstellen

  Die Prüfvorschrift soll die Architektur und die externen Schnittstellen des zu testenden Systems beschreiben.
- Beschreibung der Geschäftsprozesse

  Die Gliederung der Prüfvorschrift soll sich an den Geschäftsprozessen des zu testenden Systems orientieren.
- Beschreibung der Eingabedaten (Daten, die an das zu testende System übergeben werden)

  Die Prüfvorschrift soll beschreiben, welche Daten an das zu testende System übergeben werden können. Hieraus werden später die Testdaten abgeleitet.
- Beschreibung der Ergebnisdaten (Soll-Daten) des zu testenden Systems

  Die Prüfvorschrift soll die Berechnungsvorschrift für die Ergebnisdaten erläutern.
- Ableitung der Prüfaspekte und –ziele

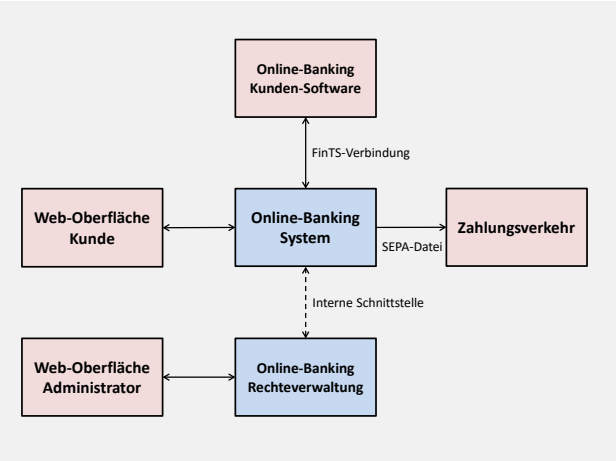
  Die Prüfvorschrift soll festlegen, was getestet werden soll.
- Nachweis der Testabdeckung

  Die Prüfvorschrift soll nachweisen, dass alle Prüfaspekte durch Tests abgedeckt sind.

Die Prüfvorschrift - Architektur & Schnittstellen

Die Prüfvorschrift beschreibt die Architektur und die Schnittstellen des betrachteten Systems mit den für das Testen relevanten Aspekten. Da Abnahmetests Black-Box-Tests sind, sind in diesem Zusammenhang nur die externen Schnittstellen des Systems von Bedeutung. Diese müssen später durch die Testumgebung simuliert werden. Dies kann manuell oder durch den Einsatz eines Tests-Tools erfolgen.

Wenn wir dies für das Bespiel Online-Banking-System betrachten, so sieht eine solche Architektur- und Schnittstellenbeschreibung (stark vereinfacht!) wie folgt aus:



Das Online-Banking-System besteht (logisch) aus der eigentlichen Online-Banking-Anwendung und einer Rechteverwaltung, in der u. a. für jeden Bankkunden das Überweisungslimit (z. B. maximal 1.000,-- € pro Transaktion und maximal 5.000,-- € täglich) vermerkt wird.

Über die Schnittstelle „Web-Oberfläche Kunde“ kann der Bankkunde auf sein Online-Banking zugreifen und z. B. Überweisungen ausführen. Über die Schnittstelle „Web-Oberfläche Administrator“ greift der Bankangestellte auf das Online-Banking-System zu und verändert z. B. das Limit für einen Kunden.

Diverse Programme zur privaten Vermögensverwaltung (z.B. WISO® oder Starmoney®) greifen über eine standardisierte Schnittstelle, die sogenannte FinTS-Schnittstelle, auf das Online-Banking-System zu.

Als Ergebnis z. B. eines Überweisungsauftrags (über die Web-Oberfläche oder über FinTS eingereicht) entsteht

eine sogenannte SEPA-Datei, die vom Online-Banking-System erzeugt und dann in den Zahlungsverkehr eingereicht wird.

Die beiden im Schaubild blau hinterlegten Komponenten *Online-Banking-System* und *Online-Banking Rechteverwaltung* bilden in unserem Beispiel gemeinsam das zu testende System.

Die Prüfvorschrift - Geschäftsprozesse

In der Prüfvorschrift werden die verschiedenen Geschäftsprozesse kurz beschrieben. Im Wesentlichen werden dabei der Ablauf und die beteiligten Schnittstellen aufgeführt.

In unserem Beispiel könnte die Beschreibung des Geschäftsprozesses „Überweisung über die Web-Oberfläche Kunde“ wie folgt aussehen:

- Über die Web-Oberfläche wird von dem Kunden eine Überweisung angestoßen.
- Das Banking-System prüft das Limit. Je nach Ergebnis der Limit-Prüfung wird die Transaktion abgewiesen oder ausgeführt.
- Bei Abweisung der Transaktion erscheint eine entsprechende Meldung auf dem Bildschirm.
- Bei Akzeptanz der Transaktion wird eine entsprechende SEPA-Datei erzeugt, und es erscheint eine entsprechende Meldung auf dem Bildschirm.

Die Prüfvorschrift - Eingabedaten

In der Prüfvorschrift werden die Eingabedaten in das zu testende System detailliert beschrieben. Für jede Schnittstelle werden alle Eingabedaten mit den folgenden Informationen spezifiziert:

- Bezeichnung
- Format
- Wert / Erläuterung

Bietet die Web-Oberfläche beispielsweise die Eingabemöglichkeit von 300 US-Dollar, so gehören hierzu die Felder „Betrag“ und „Währung“.

Bezeichnung	Format	Wert / Erläuterung
Betrag	N-1..6 oder N-1..6,N-x	Überweisungsbetrag; Die Anzahl x der Nachkommastellen hängt von der Währung ab.
Währung	Dropbox	Dropbox mit den folgenden Währungen: …

In der Prüfvorschrift wird auch die konkrete Testumgebung festgelegt, sofern diese einen Einfluss auf die Eingabedaten hat.

Bei der FinTS-Schnittstelle hängen die Eingabedaten davon ab, ob im Test eine Online-Banking Kundensoftware als Teil der Testumgebung benutzt werden soll, oder ob im Test-Tool die FinTS-Schnittstelle eigenständig implementiert wird. Im letzteren Fall kann man im Test alle Felder einer FinTS-Datei nach Belieben befüllen, daher wird auch die gesamte Datei mit allen ihren Einträgen in der Prüfvorschrift beschrieben.

Zusätzlich wird vermerkt, wie die technische Anbindung der FinTS-Schnittstelle erfolgt, welche Dateinamen zu verwenden sind, etc.

An dieser Stelle stoßen die Tester häufig auf Dokumentationslücken, insbesondere wenn es sich um ein historisch gewachsenes System handelt, das häufig migriert oder erweitert wurde.

Unklarheiten bestehen häufig darüber,

- welche Zeichen verwendet werden dürfen,
- welche Berechnungen zugrunde liegen,
- welches Format oder Codierung die Daten besitzen.

Der Tester steht nun vor der Herausforderung, unter diesen Voraussetzungen zu prüfen, ob sich das System korrekt verhält oder nicht.

Unsere Methodik schafft hier Abhilfe: In solchen unklaren Fällen werden in der Prüfvorschrift Annahmen formuliert und dokumentiert, die dem Test zu Grunde gelegt werden.

Bezeichnung	Format	Wert / Erläuterung
Verwendungszweck	ANS-0..40	Zulässige Zeichen: Keine Vorgabe: Ergänzung in der PV: alphanumerisch, "-", "/", "&","!"

Solche Festlegungen werden natürlich mit der Fachabteilung besprochen und idealerweise in die Systemdokumentation übernommen. Manchmal gelingt dies aber aus Zeitmangel in der Testphase nicht. Trotzdem ist dann das Vorgehen dokumentiert, und man kann noch zu einem späteren Zeitpunkt nachvollziehen, warum im Test beispielsweise das Zeichen "&" als zulässige Eingabe für ein Feld XYZ interpretiert wurde.

Auf diese Weise werden in der Prüfvorschrift

- die möglichen Eingangsdaten beschrieben,
- und die Abhängigkeiten zwischen den Eingangsdaten (z. B. die Anzahl der Nachkommastellen bei verschiedenen Währungen) werden deutlich.

Die Prüfvorschrift - Ergebnisdaten

Über eine oder mehrere Ergebnisschnittstellen teilt das zu testende System seine Ergebnisdaten den Partnersystemen mit. In unserem Beispiel sind dies

- die Web-Oberflächen – hier werden die Ergebnisse angezeigt,
- die FinTS-Schnittstelle – es werden Quittungsdateien erzeugt - sowie

- die Ausgabe einer SEPA-Datei für den Zahlungsverkehr.

Wie schon bei den Eingabedaten werden in der Prüfvorschrift alle in den Ergebnisschnittstellen enthaltenen Einzelwerte tabellarisch aufgeführt. In der Spalte „Wert / Erläuterung“ werden aber nicht mehr die zulässigen Eingabewerte sondern die Ergebnisse einschließlich der Berechnung angegeben, z.B.:

Bezeichnung	Format	Wert / Erläuterung
Euro-Betrag	N-12 Eurocent mit führenden Nullen	= Betrag, falls Währung = Euro; = Betrag * Kurs, falls Währung ≠ Euro, wobei Kurs = …

Auch hier kann es natürlich sein, dass manche Festlegungen nicht in der Systemdokumentation enthalten sind und die Testdokumentation Lücken schließen muss. Wie auch im Fall der Eingangsdaten wird hierbei die Definition des konkreten Sollwertes dokumentiert und begründet. Wichtig ist, dass die Berechnungsvorschrift nachvollziehbar ist.

Bei der Testfallentwicklung (s.u.) dient diese Beschreibung zur Berechnung der Ergebnisdaten (Soll-Wert). Bei der Testdurchführung kann schnell nachvollzogen werden, warum der Tester einen bestimmten Wert erwartet.

Die Prüfvorschrift - Testabdeckung & Prüfaspekte

Eine der zentralen und schwierigsten Aufgaben beim Testentwurf ist die Ermittlung der angemessenen Testabdeckung. Dabei geht es sowohl um die angemessene Auswahl der Tests (in Form von abstrakten Testfällen) als auch um eine geeignete Dokumentation der Auswahlkriterien.

Die Prüfvorschrift bietet hierzu eine wichtige Hilfestellung: Durch die Beschreibung der einzelnen Felder in jeder Ergebnisschnittstelle werden die fachlichen Verzweigungen in dem zu testenden System sichtbar.

In unserem Beispiel sahen wir die Verzweigung

- = Betrag, falls Währung = Euro;
- = Betrag * Kurs, falls Währung ≠ Euro, wobei Kurs = …

Die Analyse dieser Informationen führt zu den sogenannten **Prüfaspekten**.

Die Prüfaspekte bilden die Grundlage für die später zu entwickelnden Testfälle. Für jeden Prüfaspekt muss zunächst entschieden werden, ob er tatsächlich Gegenstand eines Testfalls werden soll oder nicht. Natürlich ist es wünschenswert, alle Prüfaspekte zu testen, - aus Budgetgründen ist dies aber nicht immer möglich oder sinnvoll.

Die Auswahl, welche Prüfaspekte durch Testfälle abgedeckt werden sollen, erfolgt im Rahmen einer Risikobewertung.

Je nach Komplexität kann diese Risikobewertung in die Prüfvorschrift integriert werden. Beispielsweise kann die Risikobewertung zu dem Schluss kommen, dass der Prüfaspekt, Transaktionen in einer Fremdwährung (≠ Euro) zu testen, nicht durch Testfälle überprüft werden muss, da aktuell nur ca. 0,01 % der Transaktionen in Fremdwährungen ausgeführt werden.

An dieser Stelle sei kurz darauf hingewiesen, dass die Prüfvorschriften regelmäßig einem Review zu unterziehen sind, ob sie noch aktuell sind. Bei zunehmender Anzahl von Transaktionen in Fremdwährung kann es notwendig werden, auch diesen Prüfaspekt abzudecken. Die Prüfvorschrift bietet den Vorteil, den aktuellen Stand der Testabdeckung zu dokumentieren und zu begründen.

In dem vorliegenden Beispiel könnten die Prüfaspekte z. B. wie folgt aussehen:

- Es soll überprüft werden, ob der Euro-Betrag korrekt berechnet wird, wenn
  - der Ausgangsbetrag in Euro vorliegt,
  - der Ausgangsbetrag in US-Dollar vorliegt.

Mit diesem Prüfaspekt wird sichergestellt, dass im Test die beiden grundsätzlich verschiedenen Fälle Ausgangswährung = Euro und Ausgangswährung ≠ Euro behandelt werden.

Der Prüfaspekt könnte auch so formuliert werden, dass Vorgaben an die Beträge und den Umrechnungskurs gemacht werden, um zu überprüfen, ob das zu testende System richtig rundet, z.B.

- Es soll überprüft werden, ob das System richtig rundet:
  - Ausgangsbetrag in US-Dollar
  - Kurs: 1 Euro = 1,0865 Dollar
  - Ausgangsbetrag = 13,50 US-Dollar
  - Euro-Betrag = 13,50 / 1,0865 = 12,42521859 → 12,43 Euro

Die Auswahl der Prüfaspekte hängt natürlich stark von der Berechnungsvorschrift ab. In manchen Fällen werden Werte einfach nur gespiegelt, oder es sind einfache Berechnungen wie in dem Beispiel. In anderen Fällen können sehr viele Werte in eine Berechnung eingehen, was dazu führt, dass die Beschreibung der Berechnungslogik sehr umfangreich ist. Entsprechend sind auch die Prüfaspekte komplexer.

Die Testspezifikation

Der zweite Pfeiler unserer Methodik ist die Testspezifikation. Dort werden aus den abstrakten Testfällen, die die Prüfvorschrift vorgibt, konkrete Testfälle abgeleitet.

Ein Testfall besteht hierbei aus einer Ablaufbeschreibung, z. B.:

- Eingabe des Limits über die Web-Oberfläche Administrator
- Eingabe der Überweisung über die Web-Oberfläche Kunde

- Entgegennahme und Prüfung der SEPA-Datei

sowie einer Festlegung aller Eingabedaten und aller Ergebnisdaten. Da es sich in der Regel um sehr viele Daten handelt, werden sogenannte *Standardbelegungen* benutzt. Für den ersten Testfall werden *alle* Eingabe- und Ergebnisdaten festgelegt. Diese Festlegung bildet dann ein Template für die folgenden Testfälle, in denen nur noch die vom Template verschiedenen Daten beschrieben werden. Ein solches Template wird als **Standardbelegung** bezeichnet. Je nach Systemausprägung kann es sinnvoll sein, eine oder mehrere solcher Standardbelegungen zu vereinbaren.

Durch die detaillierte Beschreibung der Eingabe- und der Ergebnisschnittstellen in der Prüfvorschrift wird die Erstellung der Testspezifikation sehr gut vorbereitet. Es entsteht eine Art „Kopiervorlage“ für die Testspezifikation.

Abschließend wird in einer Zuordnung *Prüfaspekt - Testfall* überprüft, ob alle Prüfaspekte durch Testfälle abgedeckt wurden.

Testdurchführung

Bei der Testdurchführung weicht die im Test geäußerte Ergebniserwartung regelmäßig von den tatsächlichen Ergebnissen ab. In diesen Fällen muss entschieden werden, wer „Recht“ hat. Da in der Prüfvorschrift die Ergebnisberechnung detailliert beschrieben ist, kann nachvollzogen werden, wie der Tester auf seine Ergebniserwartung kam. Mit der Fachabteilung kann dann geklärt werden, ob der Tester seine Vorstellungen korrigieren muss oder ob tatsächlich ein Fehler im System vorliegt.

Pflege und Wartung der Prüfvorschrift

Bei langlebigen Systemen entwickeln sich nicht nur die Systeme weiter, - auch die Testdokumentation und die Testsysteme müssen mit dieser Entwicklung Schritt halten und regelmäßig *gewartet* werden.

Während die Testspezifikation lediglich die Eingabe- und Ergebnisdaten (Soll-Wert) eines Testfalls enthält, ist die gesamte Hintergrundinformation in der Prüfvorschrift enthalten.

Bei Änderungen an dem System muss die Prüfvorschrift entsprechend angepasst werden. Aus den Änderungen an der Prüfvorschrift lassen sich dann die notwendigen Änderungen an der Testspezifikation ableiten.

Wird dieser Prozess sorgfältig eingehalten, so haben die Prüfvorschrift und die Testspezifikation dieselbe Lebensdauer wie das System selbst.

Zusammenfassung

Die *Prüfvorschrift* enthält eine Architekturbeschreibung des zu testenden Systems aus Testsicht. Ferner beschreibt es detailliert die Eingabedaten und die Berechnung der Ergebnisdaten.

Aus den Berechnungsvorschriften für die Ergebnisdaten werden die Prüfaspekte und die Testabdeckung abgeleitet.

Dieses Vorgehen hat u.a. die folgenden **Vorteile:**

- *Dokumentationslücken* des zu testenden Systems werden geschlossen.
- Die detaillierte Beschreibung der Ergebniserwartung erlaubt eine *systematische Testabdeckung* in Form von Prüfaspekten.
- Eine *Risikobewertung* bestimmt, welche Prüfaspekte durch Testfälle überprüft werden sollen.
- Die detaillierte Beschreibung der Eingabe- und der Ergebnisschnittstellen bereitet die Erstellung der Testspezifikation vor.
- Bei der Testdurchführung wird die Ergebniserwartung im Test durch die Prüfvorschrift begründet.
- Durch den Abgleich Prüfaspekt - Testfall besteht ein Nachweis der gewählten Testabdeckung.
- Die Prüfvorschrift und die Testspezifikation sind *langlebige* Dokumente. Auch nach Jahren wird zu alten Testfällen klar, warum sie durchgeführt wurden (Prüfaspekt) und wie die damals geäußerte Ergebniserwartung zu begründen ist.

Literatur

Kaner, Cem. Falk, Jack Nguyen, Hung Quoc: Testing Computer Software, Wiley & Sons, 1999.

Andreas Spillner, Tilo Linz: Basiswissen Softwaretest, Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB®-Standard, dPunkt-Verlag 2012.

IEEE-Standard 829-2008: IEEE Standard for Software and Test Documentation.

Autor

Dr. Johannes Ueberberg
Bereichsleiter Test und Qualitätssicherung
bei SRC Security Research & Consulting GmbH
johannes.ueberberg@src-gmbh.de

Ethical Software Engineering Decisions

How to make software engineering ethical

On the way from the idea to the ready-to-use software many decisions are made. These are the magical moments where the degree of quality, usefulness and ethical impact of the project and its results are determined. This article treats several practical questions about how to make software engineering ethical (if you like to do it).

Decisions in software engineering

Nothing we do is ethically neutral. Not even software engineering. Each time when we make a decision, it might have positive or negative ethical consequences. Many of them are foreseeable.

The typical points of time where we make decisions in software engineering are these:

- **Decision about whether a project takes place or not:** Both sides of the contract, the customer and the contractor, decide whether to do this project or whether to do it with this partner. If the project is set up with realistic constraints – budget, delivery date, objectives and requirements, and staffing – then it has a chance to take an ethical course. If the budget is too low or other resources are insufficient in quality or quantity, then this creates pressure and there is no correct, ethical way to reach all objectives within the given constraints. Unethical behavior usually is the practical solution in such a situation. However, budgets systematically are underestimated because the sales staff works under the pressure to win the contract. Pressure always makes people biased.
- **Decisions about requirements, architecture, development tools:** These decisions influence the resulting software's quality. Although they seem rather technical and ethically neutral, they can aggravate the problems created by unrealistic project constraints. If the budget is tight, inefficient work increases the pressure. To stay within budget, quality requirements might be neglected, technical quick-and-dirty solutions might lack sustainability and lead to ethical debt for future projects.
- **Decisions about testing and end of testing:** Testing is the last chance before delivery to detect not only defects (i.e. deviations of the software from the specified requirements) but also missing requirements, bad quality, accessibility problems, etc. However, the testers work under even higher pressure than the preceding activities because the delivery date is near, finding defects causes delays and cost for repair. Although – seen from an ethical perspective – finding defects in the software should make the team happy because it prevented them from delivering defects, usually time and budget pressure do not allow such feelings. Instead, often banana software is delivered which matures while the customer uses it.
- **Decisions about delivery:** When is software delivered? When it is in best quality and it is certain that it contains no disastrous defects? After having been tested completely, all defects being repaired? Usually not, due to time pressure. It would already be a relief to know that the software delivered has been tested completely and only uncritical defects are delivered, but usually not even this can be taken for granted. Testing complex software is costly, but it is part of the development process and should be paid by the customer.

How to consider ethics in decision-making

Decision theory has a simple model of decision-making: We decide among alternatives and evaluate them against clearly defined criteria, which might be weighted according to their importance. This decision-making process can be documented in a table like in Figure 1.

Criteria	Weight	Alternative 1	Alternative 2
Cost	1	10	1
Benefit	1	5	5
Ethical impact	1	1	10
Sum		16	16

Figure 1 – Decision table

However, if we use ethical impact – or sub-factors like safety, ergonomics, accessibility, economical sustainability – as normal decision criteria, they are in danger. If we

have a closed look at Figure 1, we see that Alternative 1 is very cheap, has an average benefit and is ethically bad. Alternative 2, however, is ethically great, average benefit, but causes high cost and therefore receives a very low rating with respect to the cost criterion. What happens is that both alternatives end with 16 points each and are evaluated as being equivalent. This happens, because this approach allows us to "buy" (or "sell") ethics. Ethics gets a price. We cannot solve this problem by giving ethical criteria a higher weight in the table. This only increases the price of ethics, but does not change the principle.

The only solution is to consider ethical criteria as a K.O. criteria. If a certain ethical level is not satisfied by a solution, it must not be considered any further. This happens when organizations to not sub-contract other organizations which do not satisfy certain ethical standards. They are not considered as partners any further and cannot make up their low ethical standards by offering cheap. If the customer is really serious about ethics!

So, this makes the difference: Ethics as a decision criterion are in danger if they are used as a normal decision criterion. Ethics must be a K.O. criterion!

Why are unethical decisions made?

Some people have decided to make unethical decisions because it is advantageous to their career. Some maybe do not even know what ethics are because in their environment, they are no topic. However, even people who are compassionate, sincere and want to lead an ethically correct life, sometimes are forced to make unethical decisions for the sake of self-protection. Either they are passively forced by constraints and pressure, knowing that if they fail to satisfy unrealistic objectives, this will have painful consequences for them. More often, however, they get actively forced by prophylactic threats and clear statements. Blackmailing and harassment can force a good person to make bad decisions, just to survive a threat that has been deliberately created by others.

In general, if a project or a whole organization is under pressure to achieve unrealistic objectives, this pressure is propagated from one person to the other, usually top-down in hierarchy.

Regularly, people get away from unsolvable ethical problems by leaving the project or the company. They then can serve as the scapegoat for those who stay. This does not solve the problem, but it seemingly relieves all others from the responsibility. One can believe that all this was one person's fault and as this person has gone, everything will be fine next time, in the next project, without changing anything, for instance the cost estimation approach which led to the unrealistic cost estimation.

Unfortunately, one can get used to unethical behavior. The first time when being involved in bribery, one might be nervous. But when then nothing happens, the deed is not discovered, then it seems to be OK and can be repeated. Corrupt people can become quite greedy then and feel invulnerable.

Rotten apples

One rotten apple can spoil the whole box. There are research results which show that people having been treated unjust recently tend to be unjust to others, too. When the unethical guys seems to win all the time, the sincere people might feel like idiots. One can live with this, knowing that one is doing right.

However, good people regularly get into the way of bad people and then these use their unfair, dangerous tricks to get what they want. You probably all know such stories. A very good book about how one rotten apple can spoil a box is Sutton's "No Asshole Rule". He uncompromisingly demands that rotten apples are not to be tolerated by any means.

What if I am a healthy apple in a rotten box?

A stable group of people consciously following the ethical path can defend their values against intruders who are less ethical. However, what happens if you find yourself in a group of people whose ethical standards are much lower than yours? Adapting to their standards can be painful, but you can also get used to being unethical. I heard that unethical behavior is much fun and makes you rich.

I do not think that we can stand group pressure a long time. A homogeneous group does not tolerate individuals who behave differently. This is independent of whether the group agrees on ethical values or on unethical values or on the color of their clothes. So, either you flee, you adapt or they destroy you.

However, usually, in normal work life, unethical people do not form a homogeneous group, but often are enemies and fight against each other. So, if you keep your head low, this can save your neck. My strategy of survival is to do everything correctly, especially when someone recommends me to do otherwise. This can be a trap. Do not do anything unethical to please them. This is a typical gang initiation procedure. It is like in criminal groups where you need to kill someone to become a member. Their knowledge of your crime gives them power over you. And sometimes, it just is a trick and they do not intend to accept you in the gang.

Ethical work thanks to software engineering practices?

But back to software engineering. How do we develop ethically correct software in an ethically correct process? The principle is simple and the tools already exist: We do everything with due diligence. Before we do something, we think about its consequences and make decisions with care.

For the decision for or against a project, all its consequences are considered, cost is estimated carefully. It is made sure that the project constraints and expectations are realistic. And we learn from our cost estimation errors to

improve continuously. We can fail once, but we should not repeat our errors.

And the same we do for the subsequent decisions: Requirements and architectural decisions are measured also by their ethical consequences. Software quality is assured by thorough testing, critical defects are removed. Problems are called “problems” and treated as early as possible.

In general, following standards and laws already brings you very near to good ethical work, because they were made for this. When you follow a standard, you always know what you do and why you do it. For specific questions concerning software engineering ethics, you can consult the ACM’s “Software Engineering Code of Ethics and Professional Practice” (see below). It starts with: “Software engineers … 1.01. Accept full responsibility for their own work.”

Bibliography

ACM, Software Engineering Code of Ethics and Professional Practice, https://www.acm.org/about/se-code

Robert I. Sutton, The No Asshole Rule: Building a Civilized Workplace and Surviving One That Isn’t, Business Plus, 2008.

Author

Andrea Herrmann

Freelance Trainer and Consultant in IT Management. 20 years of work experience in IT projects, research and training, two guest professorships. I am an associate member at IREB, speaker of the regional group of the German Informatics Society in Stuttgart.

www.herrmann-ehrlich.de

herrmann@herrmann-ehrlich.de

Haben wir das Richtige getestet?

Erfahrungen mit Test-Gap-Analyse in der Praxis

Bei langlebiger Software treten meist dort Fehler auf, wo viel geändert wurde. Testmanager versuchen daher, Änderungen besonders intensiv testen zu lassen. Unsere Studien zeigen jedoch, dass Code-Änderungen auch in gut strukturierten Testprozessen oft ungewollt ungetestet bleiben.

Test-Gap-Analyse zeigt ungetestete Änderungen auf und erlaubt Testern dadurch, sie noch rechtzeitig zu testen. Damit erlaubt Test-Gap-Analyse eine wirksame Qualitätssicherung der eigenen Test-Prozesse.

Nach einer Einführung in Test-Gap-Analyse stellen wir die Erfahrungen vor, die wir in den letzten Jahren im Einsatz bei Kunden und in der eigenen Entwicklung gesammelt haben und gehen dabei vor allem darauf ein, wie sich Test-Gap-Analyse in verschiedenen Test-Phasen einsetzen lässt.

Wie gut werden Code-Änderungen in der Praxis durch Tests wirklich abgedeckt?

In vielen Systemen machen manuelle Testfälle nach wie vor einen Großteil aller Tests aus. In einem großen System ist es alles andere als trivial, die manuellen Testfälle so auszuwählen, dass sie auch die Änderungen durchlaufen, die seit der letzten Testphase durchgeführt wurden und vermutlich die meisten Fehler enthalten.

Um besser zu verstehen, ob die Tests die Änderungen tatsächlich erreicht haben, haben wir eine wissenschaftliche Studie [1] auf einem betrieblichen Informationssystem durchgeführt. Das untersuchte System umfasst ca. 340.000 Zeilen C#-Code. Wir haben die Studie über 14 Monate Entwicklung durchgeführt und dabei zwei aufeinanderfolgende Releases untersucht.

Durch statische Analysen haben wir ermittelt, welche Code-Bereiche für die beiden Releases neu entwickelt oder verändert worden sind. Für beide Releases wurde jeweils etwa 15% des Quelltextes modifiziert. Außerdem haben wir alle Testaktivitäten erhoben. Dafür haben wir die Testüberdeckung aller automatisierten und manuellen Tests über mehrere Monate aufgezeichnet.

Eine Auswertung der Kombination aus Änderungs- und Testdaten zeigte uns, dass **etwa die Hälfte der Änderungen ungetestet in Produktion gelangten** – obwohl der Testprozess sehr systematisch geplant und durchgeführt worden war.

Welche Folgen haben ungetestete Änderungen?

Um die Konsequenzen der ungetesteten Änderungen für die Anwender des Programms zu quantifizieren, haben wir retrospektiv alle Fehler analysiert, die in den Monaten nach den Releases aufgetreten sind. Dabei zeigte sich, dass die Fehlerwahrscheinlichkeit in geändertem, ungetestetem Code fünfmal höher war, als in ungeändertem Code (und auch höher als in geändertem und getestetem Code).

Diese Studie führt uns vor Augen, dass Änderungen in der Praxis sehr häufig ungetestet in Produktion gelangen und dort den Großteil der Feldfehler verursachen. Sie zeigt uns damit aber auch einen konkreten Ansatzpunkt, um die Testqualität systematisch zu verbessern: wenn es uns gelingt, Änderungen zuverlässiger zu testen.

Warum rutscht Code durch den Test?

Die Menge an ungetestetem Code in Produktion hat uns offen gesagt überrascht, als wir diese Studie zum ersten Mal gemacht haben. Inzwischen haben wir vergleichbare Analysen in vielen Systemen, Programmiersprachen und Firmen durchgeführt und erhalten oft ein ähnliches Bild. Die Ursache für ungetestete Änderungen liegt jedoch – anders als man vielleicht vermuten könnte – nicht an mangelnder Disziplin oder am Einsatz der Tester; sondern vielmehr daran, dass es ohne geeignete Analysen sehr schwierig ist, geänderten Code in großen Systemen im Test zuverlässig zu erwischen.

Testmanager orientieren sich bei der Testauswahl häufig an den Änderungen, die im Issue-Tracker (Jira, TFS, Redmine, Bugzilla, etc.) dokumentiert sind. Für fachlich motivierte Änderungen funktioniert das erfahrungsgemäß auch oft gut. Testfälle für manuelle Tests beschreiben typischerweise Interaktionssequenzen über die Nutzeroberfläche, um gewisse fachliche Abläufe zu testen. Enthält der Issue-Tracker Änderungen einer Fachlichkeit, werden die entsprechenden fachlichen Testfälle zur Durchführung ausgewählt.

Unsere Erfahrungen zeigen jedoch, dass Issue-Tracker aus zwei Gründen keine geeigneten Informationsquellen sind, um Änderungen lückenlos zu finden. Erstens gibt es häufig technisch motivierte Änderungen, wie beispielsweise Aufräumarbeiten oder Anpassungen an neue Versionen von Bibliotheken oder Schnittstellen zu Fremdsystemen. Bei derartigen Änderungen ist es für Tester nicht nachvollziehbar, welche fachlichen Testfälle durchgeführt werden müssten, um diese technischen Änderungen zu durchlaufen.

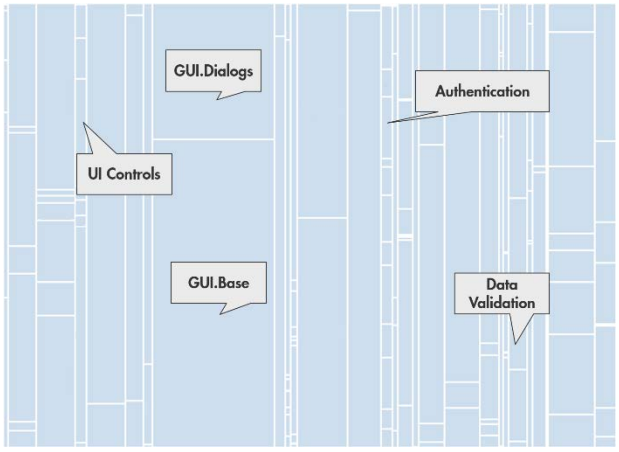


Abb. 1: Treemap mit den Komponenten des System under Test. Jedes Rechteck repräsentiert eine Komponente. Der Flächeninhalt korrespondiert mit der Größe der Komponente in Zeilen Quelltext. Exempla-risch ist die primäre Funktion einzelner Komponenten angegeben.

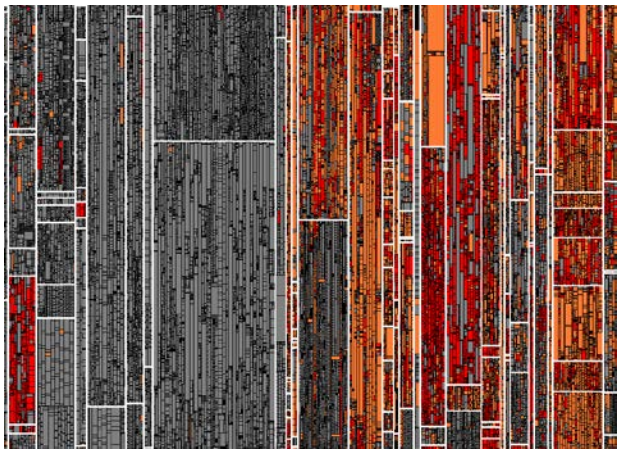


Abb. 2: Änderungen im System under Test seit dem letzten Release: Jedes kleine Rechteck stellt eine Methode im Quelltext dar. Unveränderte Methoden sind grau, neue Methoden rot und geänderte Methoden orange dargestellt.

Zweitens, und noch gravierender ist jedoch, dass in vielen Fällen zentrale Änderungen am Issue-Tracker vorbeigehen, sei es aus Zeitdruck oder aus politischen Gründen. Dadurch sind die Daten im Issue-Tracker lückenhaft. Um Änderungen lückenlos zu finden, benötigen wir daher zuverlässige Informationen, welche Änderungen im Test durchlaufen wurden und welche nicht.

Was kann man machen?

Die Test-Gap-Analyse ist ein Ansatz, der statische und dynamische Analyseverfahren kombiniert, um geänderten, aber ungetesteten Code zu identifizieren. Sie umfasst folgende Schritte:

Statische Analyse. Eine statische Analyse vergleicht den aktuellen Stand des Quelltextes des System under Test mit dem Stand des letzten Releases, um neue und geänderte Code-Bereiche zu ermitteln. Dabei ist die Analyse intelligent genug, um unterschiedliche Arten

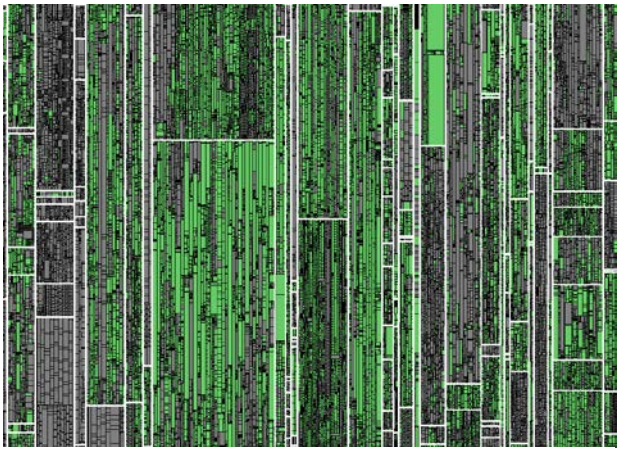


Abb.3: Testabdeckung im System under Test am Ende der Testphase: Ungetestete Methoden sind grau dargestellt, im Test durchlaufene Methoden grün.

von Änderungen voneinander zu unterscheiden. Refactorings, bei denen das Verhalten des Quelltextes nicht verändert wird (bspw. Änderung von Dokumentation, Umbenennungen von Methoden oder Verschiebungen von Code) können keine Fehler verursachen und daher herausgefiltert werden. Dadurch wird die Aufmerksamkeit auf die Änderungen gelenkt, durch die sich das Verhalten des Systems verändert hat. Die Änderungen eines der von uns analysierten Systeme sind in Abbildung 2 dargestellt. Abbildung 1 erläutert zusätzlich, wie das zugrundeliegende System aufgeteilt ist.

Dynamische Analyse. Ergänzend dazu wird mit Hilfe von dynamischen Analysen die Testüberdeckung ermittelt. Entscheidend ist dabei, dass alle durchgeführten Tests aufgezeichnet werden, also sowohl automatisierte, als auch manuell durchgeführte Testfälle. Die durchlaufenen Methoden sind in Abbildung 3 dargestellt.

Kombination. Die Test-Gap-Analyse ermittelt dann durch die Kombination der Ergebnisse der statischen und dynamischen Analysen die ungetesteten Änderungen. Abbildung 4 zeigt eine Treemap mit Ergebnissen einer Test-Gap-Analyse für dieses System. Die kleinen Rechtecke in den Komponenten repräsentieren hier die enthaltenen Methoden, ihr Flächeninhalt korrespondiert mit der Länge der Methode in Zeilen Quelltext. Die Farben der Rechtecke haben dabei die folgende Bedeutung:

- Graue Methoden wurden seit dem letzten Release nicht verändert.
- Grüne Methoden wurden verändert (oder neu programmiert) und kamen im Test zur Ausführung
- Orange (und rote) Methoden wurden verändert (oder neu programmiert) und kamen im Test **nicht** zur Ausführung.

Man kann klar erkennen, dass im rechten Bereich der Treemap ganze Komponenten mit neuem oder verändertem Code im Test bisher nicht zur Ausführung kamen. Alle darin enthaltenen Fehler können daher nicht gefunden worden sein.

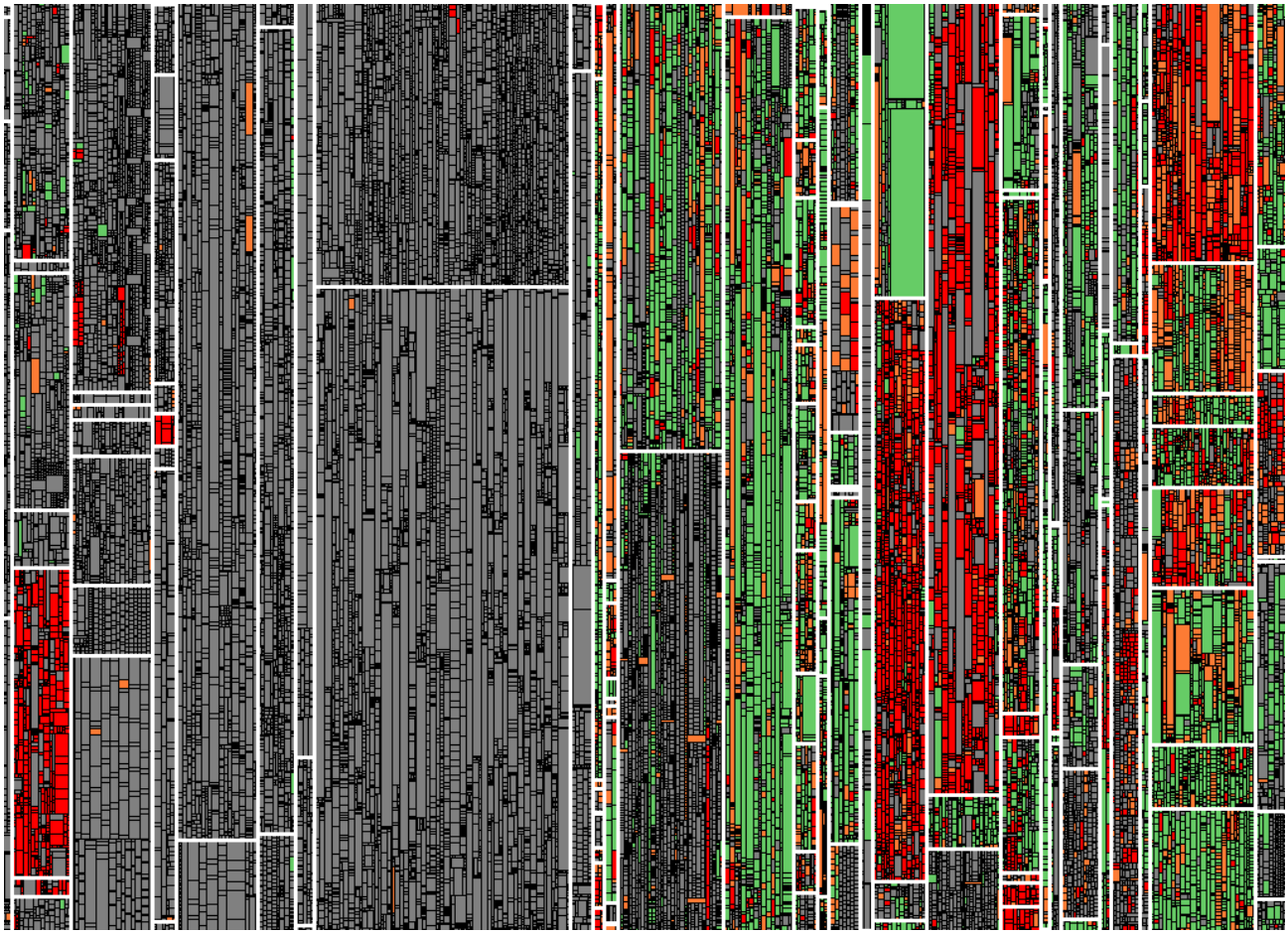


Abb. 4: Test-Gaps am Ende der Testphase. Unveränderte Methoden sind grau dargestellt. Geänderte Methoden, die getestet wurden, sind grün. Neue ungetestete Methoden sind rot, geänderte ungetestete Methoden orange dargestellt.

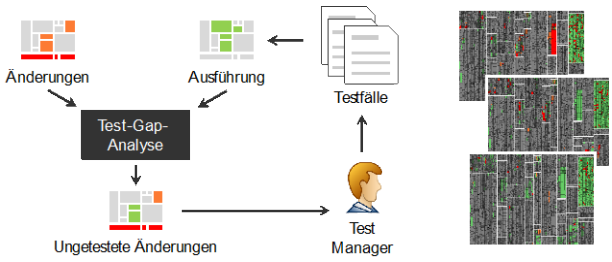


Abb. 5: Einsatz im Testprozess.

Wie kann Test-Gap-Analyse eingesetzt werden?

Nützlich wird die Test-Gap-Analyse dann, wenn sie kontinuierlich ausgeführt wird, beispielsweise jede Nacht, um morgens einen Überblick über die ausgeführten Tests und Änderungen bis zum letzten Abend zu geben. Hierfür werden Dashboards mit Informationen zu den Test-Gaps erstellt, wie in Abbildung 5 gezeigt.

Die Dashboards erlauben den Test-Managern, rechtzeitig zu entscheiden, ob weitere Testfälle notwendig sind, um die verbleibenden Änderungen noch während der Test-Phase zu durchlaufen. Ob das gelungen ist, kann am nächsten Tag in den neu berechneten Dashboards abgelesen werden.

Wenn mehrere Testumgebungen parallel betrieben werden, sollte für jede ein eigenes Dashboard eingerichtet werden, um die Test-Coverage gezielt zuordnen zu können. Zusätzlich gibt es ein Dashboard, in dem die Informationen aus allen Umgebungen zusammenlaufen. In Abbildung 6 ist ein Beispiel mit drei verschiedenen Test-Umgebungen dargestellt:

- **Test**. In dieser Umgebung führen Tester ihre manuellen Testfälle durch.
- **Dev**. In dieser Umgebung werden die automatisierten Testfälle durchgeführt.
- **UAT**. In der User-Acceptance-Test Umgebung führen Endanwender explorative Tests mit dem System under Test durch.
- **All**. Führt die Ergebnisse der drei Testumgebungen zusammen.

Für welche Projekte ist Test-Gap-Analyse einsetzbar?

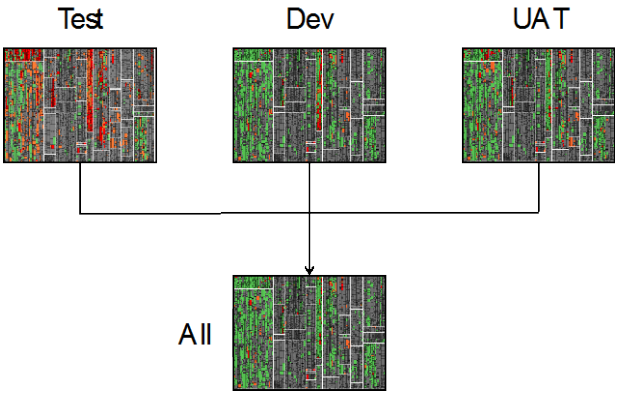


Abb. 6: Einsatz von mehreren Dashboards für detaillierte Analyse.

Wir haben Test-Gap-Analyse bereits in den unterschiedlichsten Projekten eingesetzt: von betrieblichen Informationssystemen bis hin zu eingebetteter Software, von C/C++ über Java, C# und Python bis hin zu ABAP. Faktoren, die die Komplexität der Einführung beeinflussen, umfassen unter anderem:

- **Ausführungsumgebung**. Virtuelle Maschinen (z.B. Java, C#, ABAP) erleichtern die Erhebung von Test-Coverage-Daten.
- **Architektur**. Bei Server-basierten Anwendungen müssen die Test-Coverage-Daten auf weniger Maschinen erhoben werden, als bei Fat-Client Anwendungen.
- **Testprozess**. Definierte Testphasen erleichtern die Planung und Begleitung.

Was bringt Test-Gap-Analyse im Hotfix-Test?

Beim Test von Hot-Fixes steht meist nur sehr wenig Zeit zur Verfügung. Ziele im Hotfix-Test sind einerseits sicherzustellen, dass der behobene Fehler nicht mehr auftritt und andererseits, dass dabei keine neuen Fehler eingebaut wurden. Für letzteres sollte wenigstens sichergestellt werden, dass alle im Hot-Fix durchgeführten Änderungen durchlaufen wurden. Hierfür wird in der Test-Gap-Analyse der Release-Stand als Referenzversion definiert und alle Änderungen ermittelt, die für das Hot-Fix (bspw. auf einem eigenen Branch) durchgeführt wurden, wie in Abbildung 7 dargestellt.

Mit Hilfe der Test-Gap-Analyse wird dann ermittelt, ob im Fehlernachtest tatsächlich alle Änderungen durchlaufen worden sind. Im Beispiel in Abbildung 8 zeigt sich, dass ein Teil der Methoden noch ungetestet ist. Unsere Erfahrungen haben gezeigt, dass sich gerade in Hot-Fix-Tests durch Test-Gap-Analyse leichtgewichtig und einfach die Sicherheit erhöhen lässt, durch die Änderungen keine neuen Fehler einzubauen.

Was bringt Test-Gap-Analyse im Release-Test?

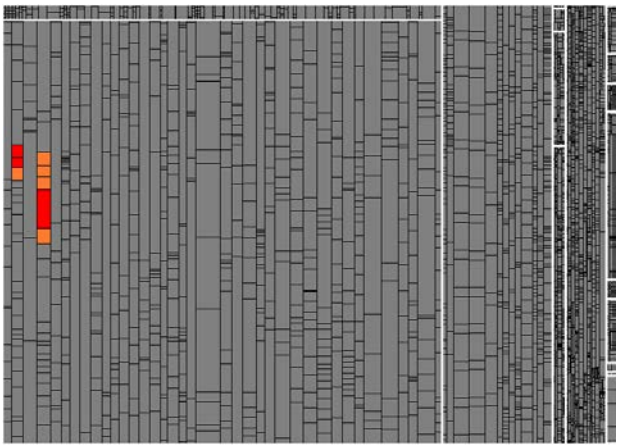


Abb. 7: Im Zuge eines Hot-Fix geänderte Methoden.

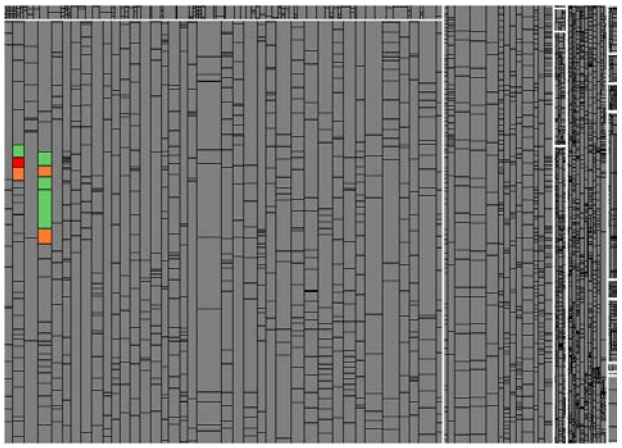


Abb. 8: Beim Fehlernachtest eines Hot-Fix getestete und ungetestete Methoden.

Als Release-Test bezeichnen wir in diesem Artikel die Test-Phase vor einem größeren Release, in der typischerweise sowohl neu implementierte Funktionalität überprüft, als auch Regressionstests durchgeführt werden. Häufig kommen dabei unterschiedliche Arten von Tests zum Einsatz.

Unsere Erfahrung hat gezeigt, dass der Einsatz der Test-Gap-Analyse die Menge an Änderungen, die ungetestet in Produktion gelangen, deutlich reduziert.

In Abbildung 9 ist eine Test-Gap-Treemap des gleichen Systems dargestellt, das auch in Abbildung 4 abgebildet ist. Während Abbildung 4 retrospektiv ermittelt wurde, ist Abbildung 9 ein Snapshot aus einer Iteration, in der Test-Gap-Analyse kontinuierlich eingesetzt wird. Dabei werden sowohl manuelle, als auch automatisierte Tests betrachtet. Es ist klar zu erkennen, dass es deutlich weniger Test-Gaps gibt.

Unsere Beobachtung ist auch, dass in vielen Fällen einige Test-Gaps bewusst in Kauf genommen werden, bspw. weil der zugehörige Quelltext über die Benutzeroberfläche noch nicht erreichbar ist. Entscheidend ist je-

doch, dass es sich hierbei um bewusste, fundierte Entscheidungen handelt, deren Auswirkungen abschätzbar sind.

Wo sind die Grenzen von Test-Gap-Analyse?

Wie jedes Analyseverfahren hat auch die Test-Gap-Analyse Grenzen. Ihre Kenntnis ist entscheidend, um sie sinnvoll einsetzen zu können.

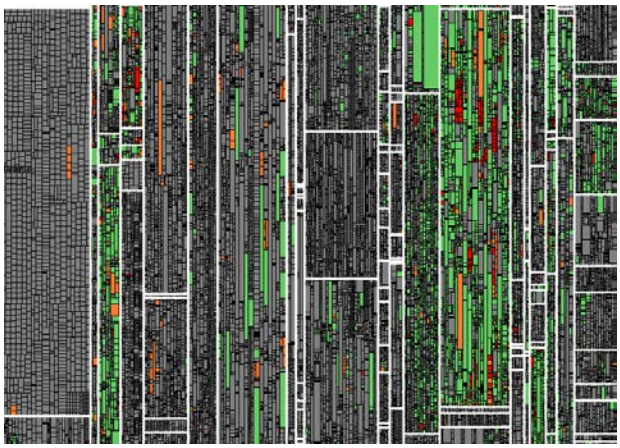


Abb. 9: Weniger Test-Gaps bei kontinuierlichem Einsatz.

Eine Grenze von Test-Gap-Analyse sind Änderungen, die auf Konfigurationsebene durchgeführt werden, ohne dass dabei Code verändert wird, da sie dadurch der Analyse verborgen bleiben.

Eine weitere Einschränkung von Test-Gap-Analyse ist die Aussagekraft von durchlaufenem Code. Test-Gap-Analyse betrachtet, welcher Code beim Test zur Ausführung gekommen ist. Wie gründlich bei der Ausführung getestet wurde, bleibt der Analyse verborgen. Dadurch ist es prinzipiell möglich, dass Fehler unentdeckt bleiben, obwohl der durchlaufene Code von der Analyse als "grün" dargestellt wird. Dieser Effekt wird größer, je gröber die Messung der Code-Coverage ist.

Der Umkehrschluss gilt jedoch: Roter und orangener Code ist nicht durchlaufen worden. Enthaltene Fehler können daher nicht gefunden worden sein.

Unsere Erfahrung aus der Praxis zeigt, dass die Lücken die durch den Einsatz von Test-Gap-Analyse aufgedeckt werden meist so groß sind, dass substantielle Erkenntnisse über Schwächen im Test-Prozess aufgedeckt werden. Bei diesen großen Lücken kommen die oben beschriebenen Grenzen nicht zum Tragen.

Ausblick

Ein weiteres spannendes Anwendungsfeld der beschriebenen Analysetechniken ist der Einsatz in einer Produktionsumgebung. Die aufgezeichneten Ausführungen sind dabei nicht mehr die ausgeführten Testfälle, sondern die Systeminteraktionen durch die Endanwender. Dadurch lässt sich ermitteln, welche der Features, die im letzten Release eingebaut wurden, eigentlich durch die Anwender verwendet werden. In unserer Erfahrung ergeben sich dabei oft Überraschungen.

In Abbildung 10 ist die Nutzung eines betrieblichen Informationssystems in Anlehnung an einen Gantt-Chart dargestellt [2]. Jede Zeile repräsentiert ein Feature der Anwendung. Auf der X-Achse ist der Messzeitraum dargestellt. An Wochenenden und in der Weihnachtszeit ist erwartungsgemäß weniger Nutzung, als in den anderen betrachteten Tagen. In der Abbildung sind jedoch

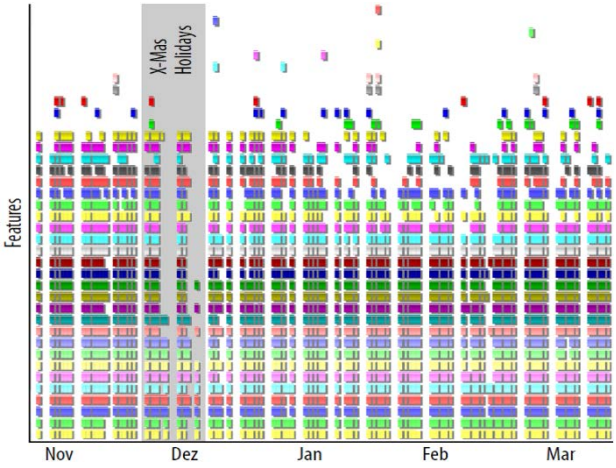


Abb. 10: Nutzung der Features eines betrieblichen Informationssystems in Produktion.

nur die Features dargestellt, die überhaupt verwendet wurden. Die Analyse hat aber gezeigt, dass 28% der Features der Anwendung gar nicht verwendet wurden, was für alle Beteiligten Stakeholder unerwartet war. In diesem Fall hat die Analyse zur Löschung von etwa einem Viertel des Quelltextes der Anwendung geführt, so dass sich in den folgenden Releases eine Reihe der Testaufwände einsparen oder nutzbringender einsetzen ließen[1].

Weitere Informationen

Wir haben unter **www.testgap.io** weiterführende Materialien zur Test-Gap-Analyse zusammengestellt, u.a. Forschungsarbeiten, Blog-Einträge und Werkzeugunterstützung. Darüber hinaus freuen wir uns als Autoren auch per Email über Fragen und Feedback (auch kritisches) zum Artikel oder zu Test-Gap-Analyse allgemein.

[1] Für Nutzungsanalyse auf Feature-Ebene sind einige Techniken erforderlich, die über die in diesem Artikel beschriebenen Methoden hinausgehen. Sie sind im referenzierten Paper beschrieben.

Literatur

[1] **S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K. Prommer.** *Did we test our changes? assessing alignment between tests and development in practice.* In Proceedings of the Eighth International Workshop on Automation of Software Test (AST'13), 2013.

[2] **E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer.** *Feature profiling for evolving systems.* In Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, pages 171–180. IEEE, 2011.

Autoren

Dr. Elmar Jürgens

Elmar Jürgens hat für seine Doktorarbeit über Qualitätsanalysen den Software-Engineering-Preis der Ernst-Denert-Stiftung erhalten. Er ist Mitgründer der CQSE GmbH und begleitet Teams bei der Verbesserung ihrer Qualitätssicherungs- und Testprozesse. Er wurde 2015 zum Junior-Fellow der GI ernannt.

juergens@cqse.eu

Dr. Dennis Pagano

Dennis Pagano hat in Software Engineering promoviert und begleitet als Berater für Software-Qualität bei der CQSE GmbH viele Firmen beim Verbessern ihrer Testprozesse. Er ist aktiv an der Entwicklung der Test-Gap-Analyse beteiligt. Seine Forschung wurde u.a. mit einem Distinguished Paper Award auf der MSR ausgezeichnet.

pagano@cqse.eu