
PORSCHE
INFORMATIK GMBH
Behavior-Driven Development
In der Praxis

Vorstellung

- **Manfred Hantschel**
- Softwareentwickler bei Porsche Informatik GmbH in Salzburg
- gut 12 Jahre in Softwareentwicklung tätig
- Aktuell: Technische Projektverantwortung für Neuwagenverwertungsplattform
- Einführung von BDD in mehreren Projekten

Dieser Vortrag konzentriert sich auf folgende zwei Punkte:

1. Was ist BDD (Behavior-Driven Development) eigentlich?
    - Was sind die Vorteile und Ziele von BDD?
    - Woraus ist BDD entstanden?
    - In welchem Bereich der Entwicklung ist die Vorgehensweise angesiedelt?
    - Wie wird BDD umgesetzt?

2. Wie funktioniert die Umsetzung von BDD in der Praxis?
    - Wie sieht der typische Entwicklungsprozess aus?
    - Wie werden Stories verarbeitet?
    - Wie werden Tests umgesetzt?
    - Woraus lassen sich Vorteile ziehen?

Entstanden ist BDD aus ursprünglich aus dem Test-Driven Development. Es wurde zum ersten Mal 2006 von Dan North beschrieben.

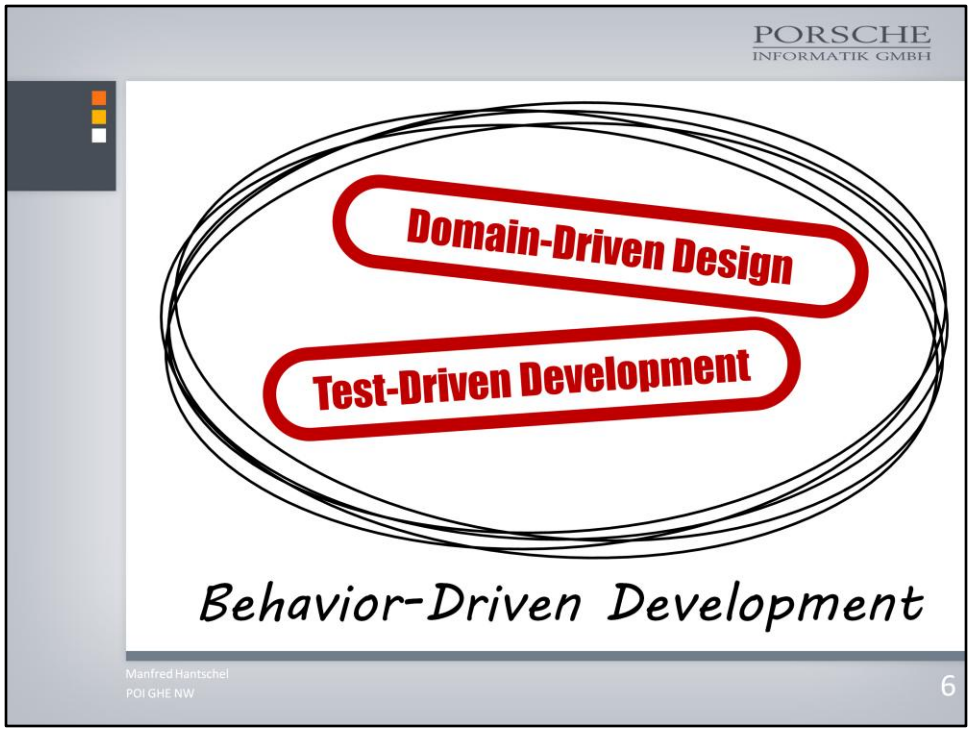
Er bemerkte, dass TDD oft nicht erfolgreich war. TDD wurde zwar umgesetzt, aber eher als Last empfunden, da TDD nicht in den Designprozess integriert wurde.

BDD versucht nun die Probleme zu lösen und geht dabei einen guten Schritt weiter.

In erster Linie soll die Zusammenarbeit zwischen

- der Qualitätssicherung
- der Entwicklungsabteilung
- und dem Kunden, bzw. Product Owner

verbessert werden. Aus dieser Zusammenarbeit soll dann TDD als Benefit entstehen.



Behavior-Driven Development versucht die Prinzipien von Domain-Driven Design mit denen von Test-Driven Development zu vereinen.

Domain-Driven Design legt den Schwerpunkt der Softwareentwicklung auf die Fachlogik der Anwendung anstatt auf die technische Umsetzung. Die komplexen fachlichen Zusammenhänge werden zunächst in einer Geschäftslogikschicht abgebildet und erst dann werden die Teilbereiche umgesetzt.

In der Praxis bewährt sich der DDD Ansatz im höchsten Maße. Durch die Schichtenarchitektur generell wird die Problemlösung vereinfacht und die Arbeitsteilung erleichtert. Leider laufen Entwickler aber auch Gefahr die eigentliche Aufgabe aus den Augen zu verlieren. Die Umsetzung ist toll, nur hilft sie dem Benutzer nicht.

Test-Driven Development soll ja den Entwickler dazu anhalten einen evolutionären Prozess in der Entwicklung einzuhalten. Es soll den Entwickler dazu bringen, sich in erster Linie um das Problem selbst zu kümmern, und nicht nur um die Umsetzung einer Lösung. Dazu soll der Entwickler vor der Umsetzung zuerst Tests dazu schreiben.

In der Praxis ist das leider nicht so einfach. Unit-Tests fokussieren den Entwickler schon sehr stark auf die Umsetzung der Lösung. Konzentriert sich der Entwickler zuerst auf Unit-Tests wird er schon frühzeitig mit Problemen konfrontiert, die technische Entscheidungen fordern. Mit Integrations-Tests verhält es sich etwas besser, jedoch decken diese meist nur die Serviceschicht ab ohne auf Details der eigentlichen Aufgabenstellung einzugehen.

Und wie auch für Domain-Driven Design, gilt auf für Behavior-Driven Development: BDD ist nicht nur eine Technik, BDD ist eine Methode der Softwareentwicklung.

Für das Domain-Driven Design stellt BDD eine Sprache zur Verfügung, mit der es möglich ist die fachlichen Abläufe detailliert und standardisiert zu formulieren. Mit Hilfe der Sprache lassen sich UserStories verfassen und Akzeptanzkriterien definieren.

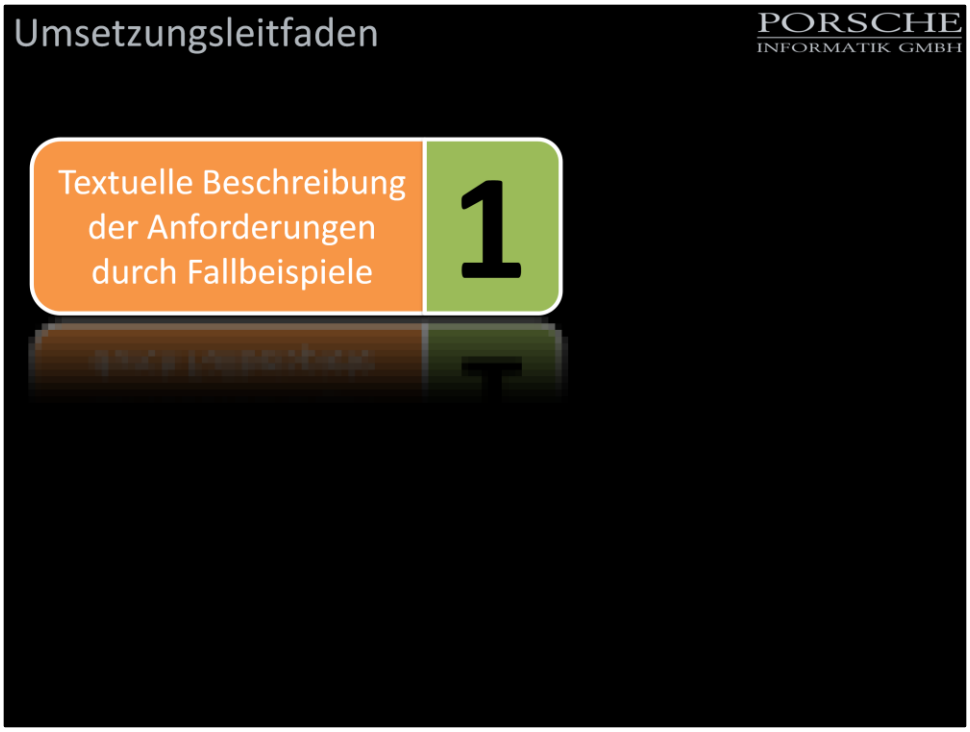
Die Sprache orientiert sich dabei am Benutzer selbst und ist für alle Beteiligten verständlich. Bei der Beschreibung wird gänzlich auf technische Feinheiten verzichtet, lediglich dem fachlichen Ablauf wird ein logischer Prozess zugrunde gelegt.

Und bereits da werden viele Probleme gelöst. Aus meiner Erfahrung tauchen hier oft schon die großen Probleme der Umsetzung auf – nämlich dann wenn der fachliche Prozess schon nicht logisch ausgereift ist. Die Entwicklung kann fast alles umsetzen, einzige Vorbedingung: die Aufgabe muss deterministisch sein. Für die Formulierung des Akzeptanztests muss das Problem auf einem Level beleuchtet werden, das logische Probleme frühzeitig zu Tage fördert.

Für das Test-Driven Development stellt BDD ein Werkzeug zur Verfügung - nämlich Akzeptanztests.

Akzeptanztests konzentrieren sich, wie von TDD gefordert, auf das Problem selbst. Die Umsetzung der Lösung kann zunächst vollkommen ignoriert werden. Die Tests bewegen sich auf einer rein fachlichen Ebene, anders als Unit-Tests. Trotzdem ist es möglich, ja gefordert, auf fachliche Details einzugehen.

Durch die Schlüsselwörter ist es möglich die Akzeptanztests automatisch zu verarbeiten und programmtechnisch Tests auf Grundlage der Kriterien verfassen.

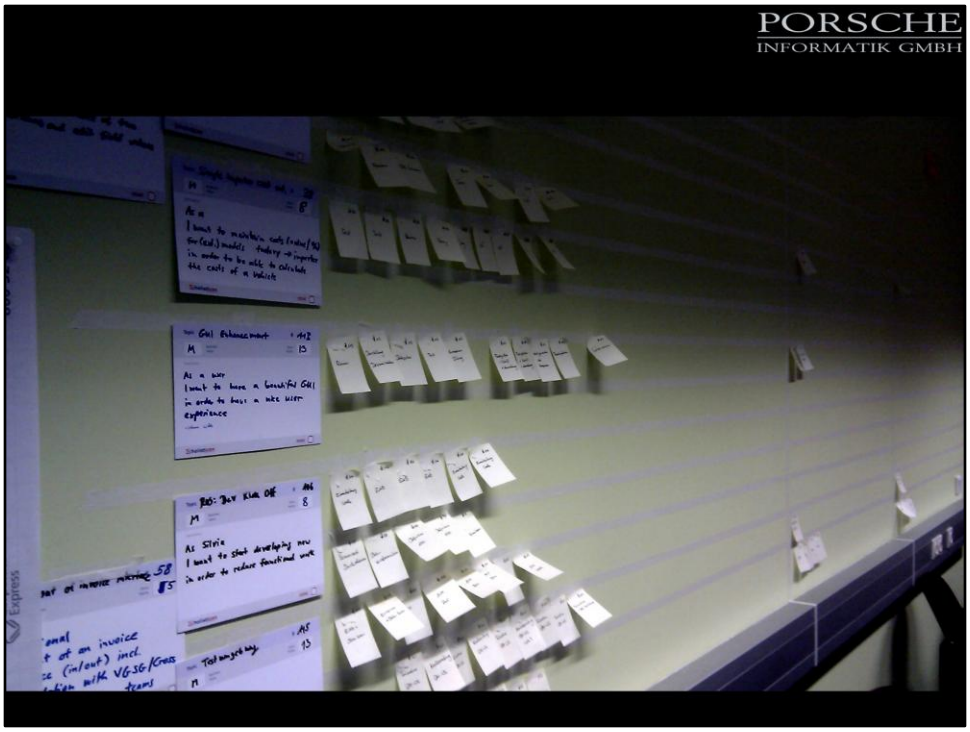


Wie kann man sich nun vorstellen, dass BDD im Entwicklungsprozess umgesetzt werden kann?

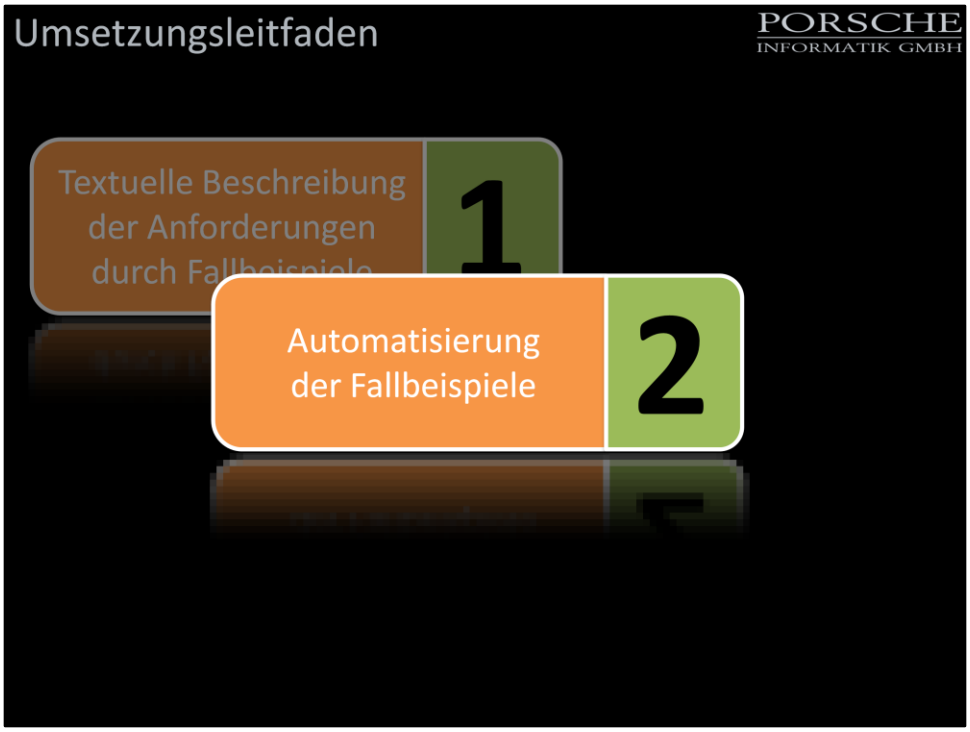
Zunächst werden die Anforderungen textuell durch Fallbeispiele beschrieben.

Fallbeispiele haben einen anderen Fokus als Use Cases, Konzepte, Protokolle, Lasten- oder Pflichtenhefte: Fallbeispiele beschreiben genau einen Anwendungsfall aus der Sicht eines Benutzers. Wir schreiben Applikationen für Benutzer und darauf liegt der Fokus.

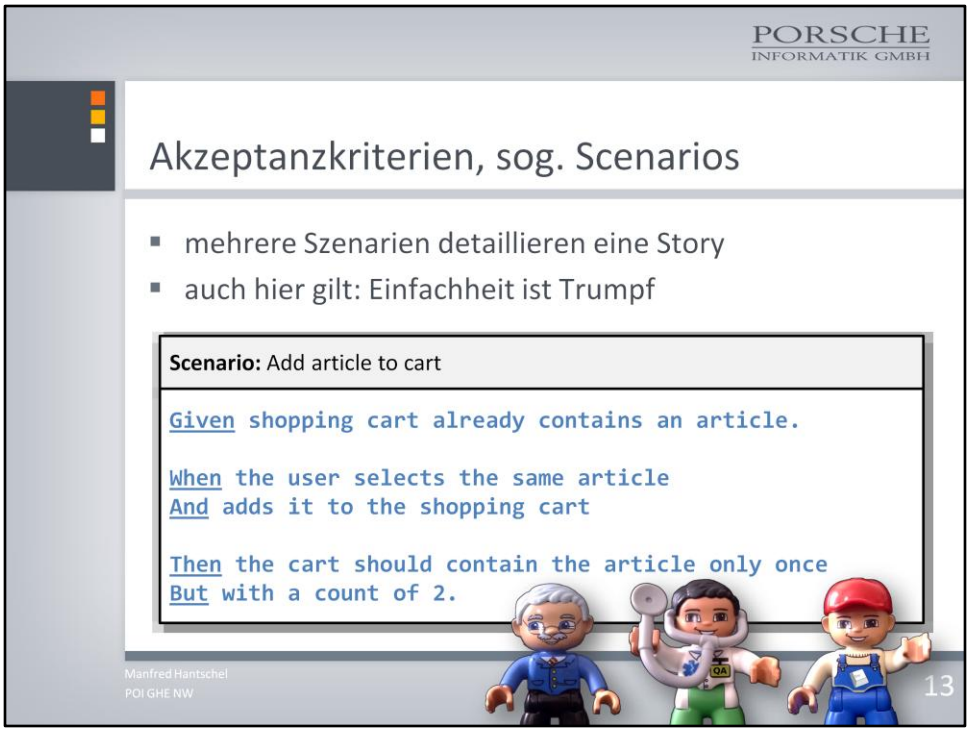
Klar, wir können auch Lasten- und Pflichtenhefte scheiben, aber die sind für Kunden. Wir können auch UseCases scheiben, aber die sind für Tester. Wir können auch Konzepte schreiben, aber die sind für Entwickler. Unser Ziel ist eine Sprache zu sprechen, die alle verstehen, und daher schreiben wir User Stories.



Und weil wir agil entwickeln, brauchen wir die User Stories sowieso, nämlich als Sprintziele. Aber erst durch BDD finden sie User Stories wirklich Einzug in unser Projekt, da sie maschinell verarbeitet werden müssen.



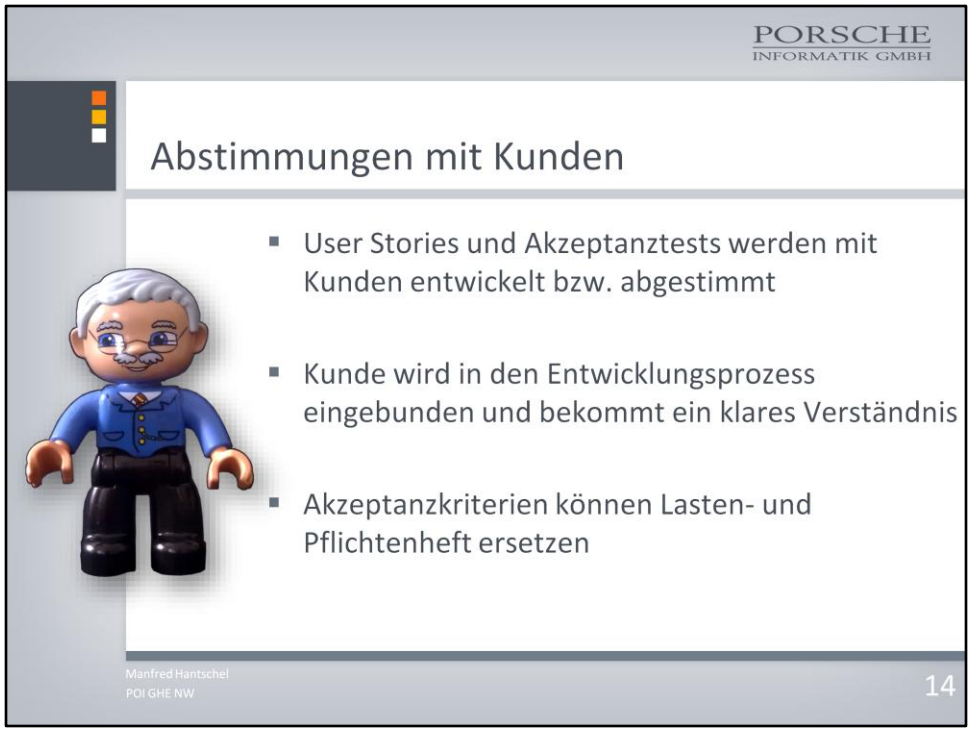
Als nächster Schritt werden die Fallbeispiele um Akzeptanzkriterien erweitert und automatisiert.



Die User Stories werden um Akzeptanzkriterien erweitert. Zu jeder User Story können natürlich mehrere Akzeptanzkriterien verfasst werden. Diese Scenarios detaillieren die Story, beschreiben Sonder- und Problemfälle und listen mögliche Aktionen des Benutzers auf.

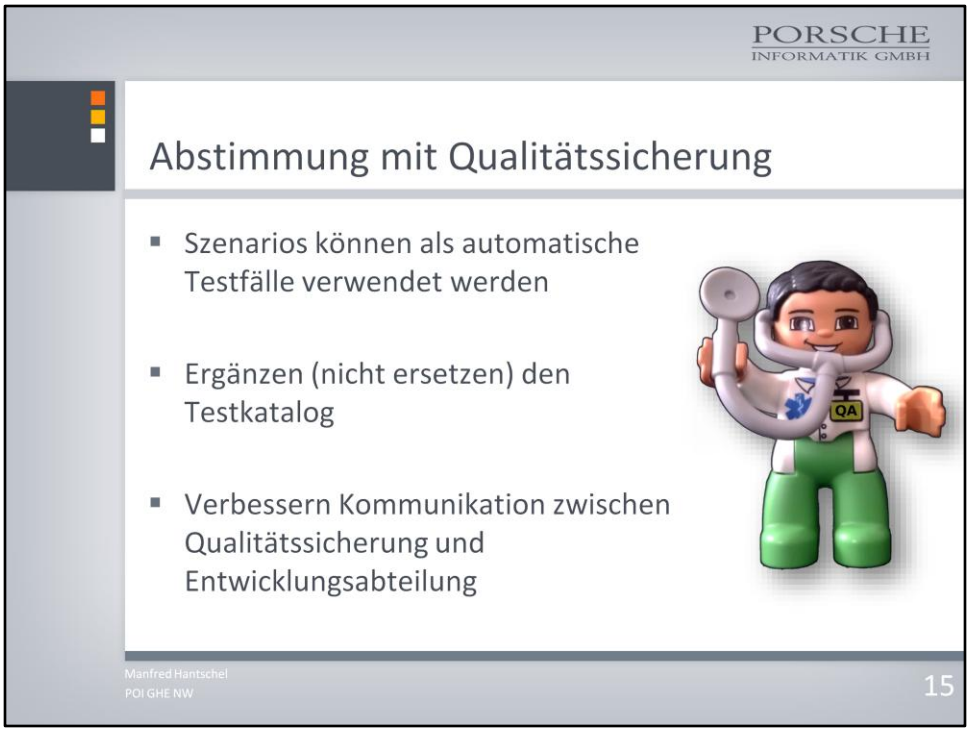
Aus diese User Stories sind in einer Sprache verfasst, die alle Beteiligten lesen und verstehen können.

Die Texte basieren auf Schlüsselwörtern, im Normalfall „Given“, „When“ und „Then“. Um den Text besser lesbar zu gestalten bieten machen Frameworks auch andere Schlüsselwörter wie z.B. „And“ und „But“ an. Ebenso bieten die Frameworks das Parsen von Variablen und Tabellen aus den Texten selbst an. Der restliche Text wird 1:1 an das Testframework übergeben.



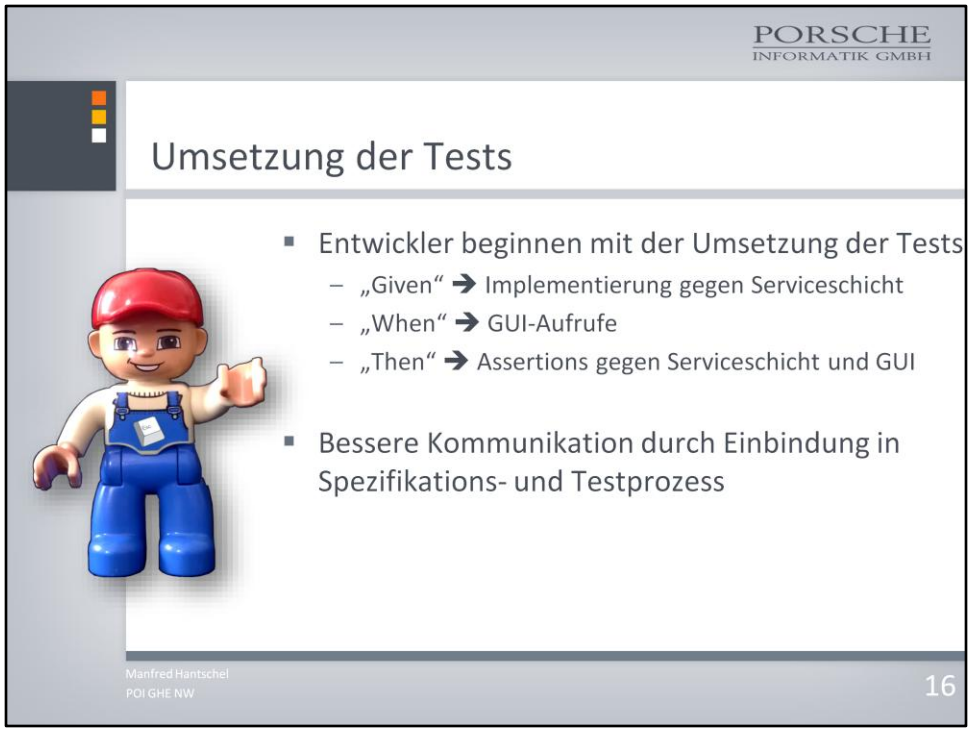
Die fertigen User Stories und Akzeptanzkriterien werden mit dem Kunden gemeinsam entwickelt, oder zumindest mit ihm abgestimmt. Der Kunde wird somit in den Entwicklungsprozess mit eingebunden. Hier zeigt sich schon ein erster Vorteil. Früher sind fachliche Fehler erst im fertigen Produkt aufgetaucht. Die Stories geben dem Entwicklungsteam ein Werkzeug in die Hand, mit dem der fachliche Ablauf vollständig definiert werden kann.

Wenn man wirklich agil arbeitet, können die Akzeptanzkriterien ein Lasten- und Pflichtenheft zumindest teilweise ersetzen.

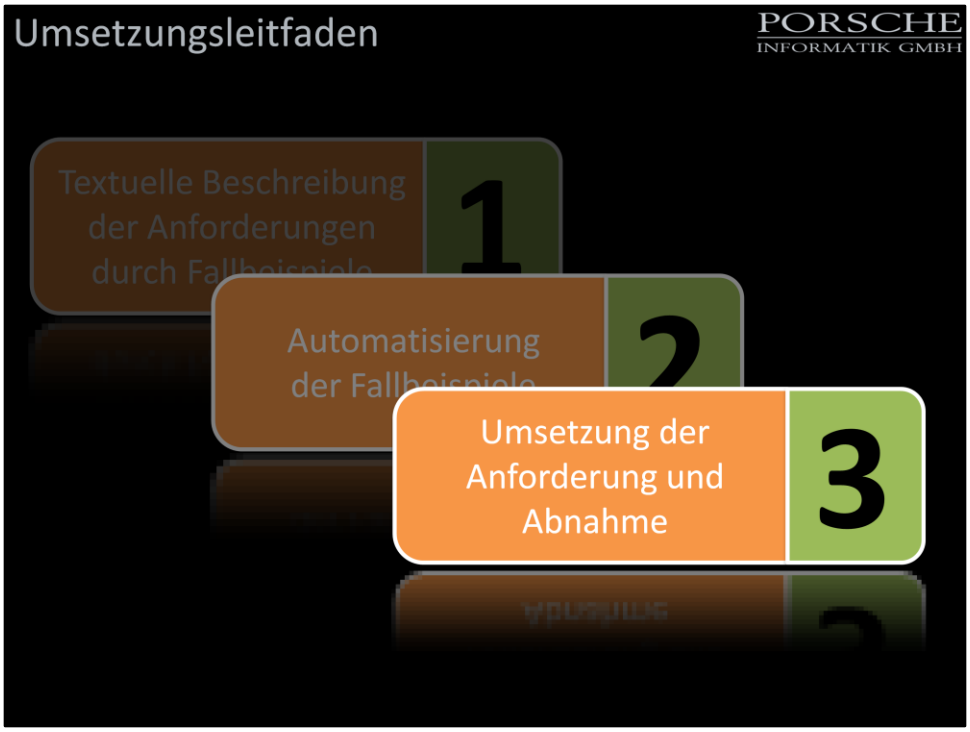


Auch die Qualitätssicherung wird frühzeitig in den Prozess eingebunden. Schon das ist ein enormer Vorteil. Oft wird die Qualitätssicherung in Projekten stiefmütterlich behandelt. Die fertige Applikation wird übergeben und die Qualitätssicherung soll schauen wie sie damit zurecht kommt.

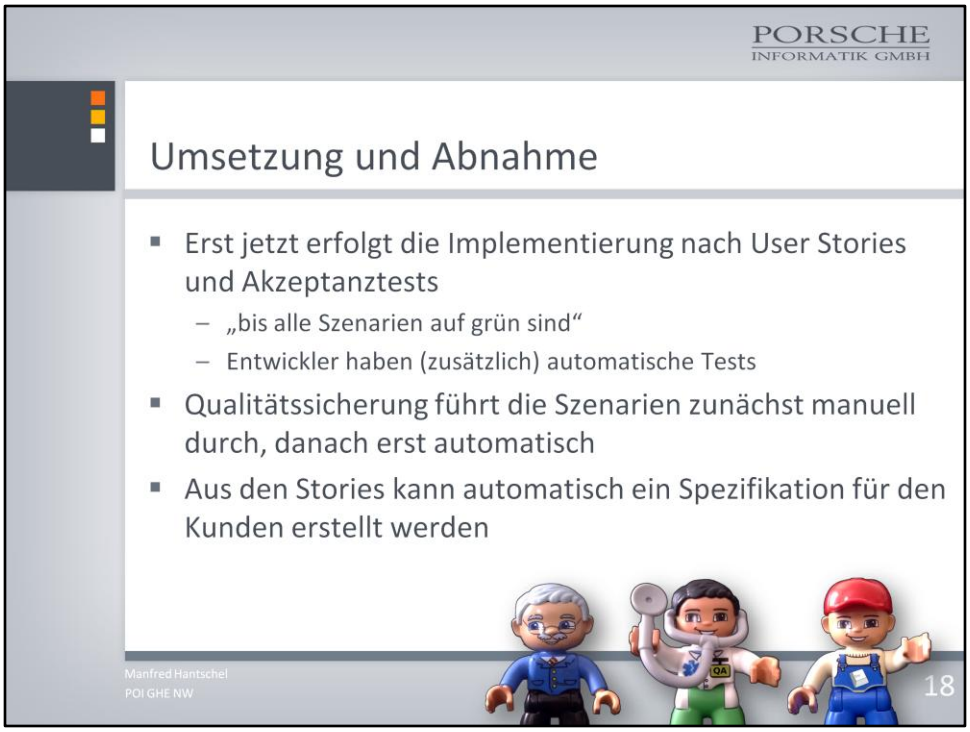
Da die Akzeptanzkriterien den Testkatalog ergänzen wird der Qualitätssicherung auch viel Arbeit abgenommen. GUI-Tests werden mit dem Produkt mitgeliefert und muss nicht nachträglich und umständlich definiert werden.



Die Entwickler kümmern sich zunächst um die Umsetzung der Tests. Bevor die Applikation fertig ist, könnte sie theoretisch schon getestet werden. Auch hier zeigt sich der Vorteil, dass Entwickler schon frühzeitig in den Spezifikationsprozess eingebunden werden.



Erst jetzt folgt die Umsetzung der Anforderung.



Erst im dritten Schritt wird die Anforderung umgesetzt. Das Entwicklungsteam hat bereits automatische Tests an der Hand.

Die Qualitätssicherung für die Test zunächst manuell durch, bis klar ist, dass die Szenarien den nötigen Testumfang abdecken. Danach können diese Tests jederzeit automatisch durchgeführt werden.

Als zusätzlicher Benefit kann aus den User Stories auch automatisch eine Spezifikation der Applikation abgeleitet werden.

Dieser Vortrag konzentriert sich auf folgende zwei Punkte:

1. Was ist BDD (Behavior-Driven Development) eigentlich?

2. Wie funktioniert die Umsetzung von BDD in der Praxis?
    - Wie sieht der typische Entwicklungsprozess aus?
    - Wie werden Stories verarbeitet?
    - Wie werden Tests umgesetzt?
    - Woraus lassen sich Vorteile ziehen?

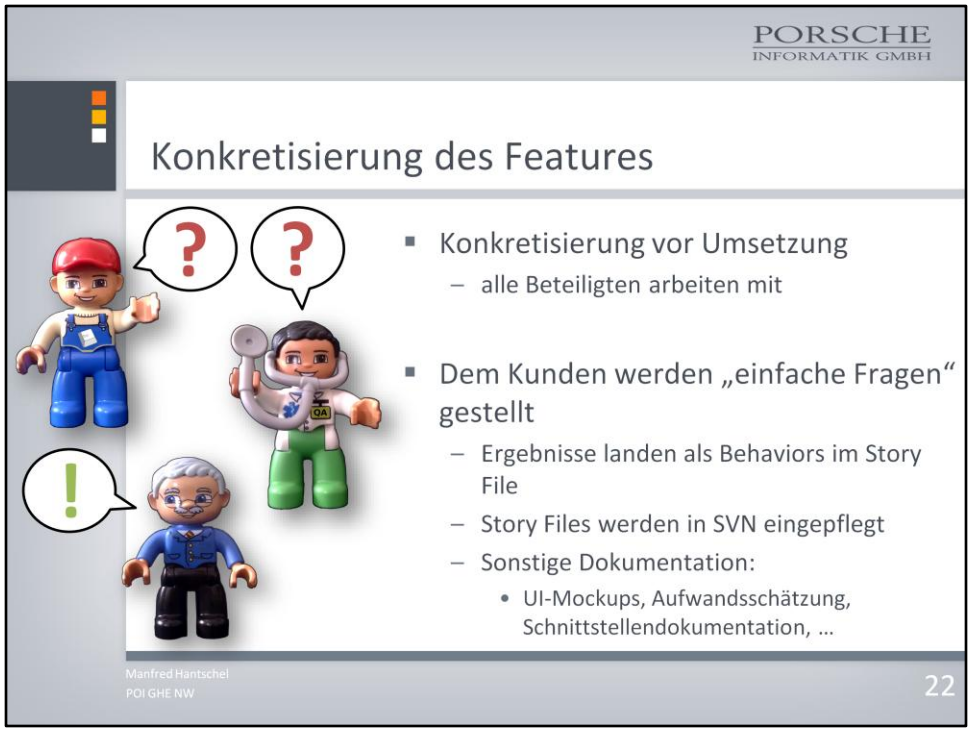
Porsche Informatik GmbH

- Sitz in Salzburg, Teil der Porsche Holding, 100% Tochter der Volkswagen AG
  - http://porscheinformatik.com
- ca. 250 Mitarbeiter in allen Bereichen der Informationstechnologie
- mehr als 30 verschiedene Lösungen für Kfz-Einzel- und Großhandel, viele davon international
- BDD wird z.Z. in drei großen Entwicklungsprojekten umgesetzt

Am Anfang war die Idee. Diese wird meist vom Kunden bzw. dem Product-Owner ausgebrütet und formuliert. Die Idee kann aber auch von anderen Beteiligten kommen, z.B. von der Entwicklung. Die Idee wird bei uns in einer Anforderungsdatenbank, noch in Prosa, erfasst. Die Anforderung enthält eine grobe Vorstellung wie das Feature aussehen soll und beschreibt Hintergrundinformationen.

In der Porsche Informatik wird die Anforderungsdatenbank auch für die offizielle Statusverfolgung genutzt, da sich unsere Kunden mit dem agilen Vorgehen oft noch nicht richtig anfreunden können. Teilweise dient die Anforderungsdatenbank auch als Quelle für Aufwandsschätzungen, besonders wenn die Umsetzung für das nächste Wirtschaftsjahr geplant ist. Auch das deckt sich noch nicht mit dem agilen Vorgehen.

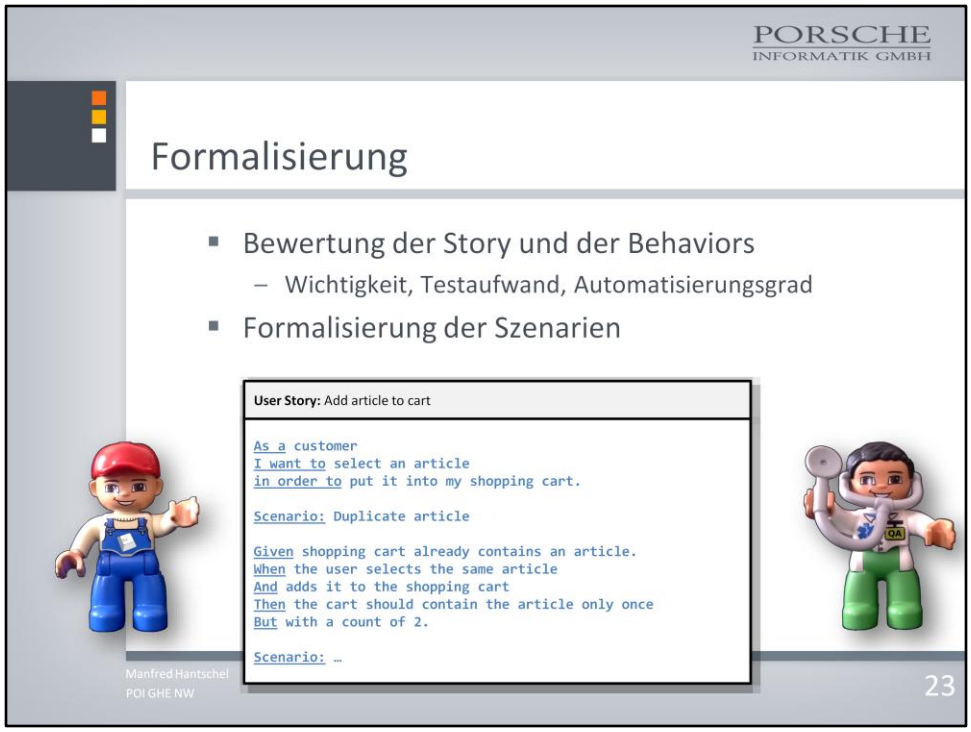
Für große Anforderungen, die nicht in einem Sprint umgesetzt werden können, werden jetzt User Epics verfasst. User Epics sind Sammlungen von User Stories, es müssen aber zu Beginn nicht alle User Stories ausformuliert sein.



Bevor das Feature umgesetzt wird, wird es noch konkretisiert. Dies geschieht durch Kunde, QA und Entwicklung gemeinsam.

Grundsätzlich werden dem Kunden „einfache Fragen“ gestellt. Die Ergebnisse aus diesen Abstimmungen werden einem Story File festgehalten, in Behaviors aufgeteilt und im Versionskontrollsystem (SVN) eingepflegt. Behaviors entsprechen den Szenarien, müssen jedoch noch nicht so formuliert sein, dass sie maschinell verarbeitet werden können.

Je nach Umfang des Features entstehen auch noch zusätzliche Dokumentation: UI-Mockups, eine Aufwandsschätzungen, Schnittstellendokumentationen, usw. Wir achten darauf, dass jede einzelne zusätzliche Dokumentation auch in der Story referenziert ist.

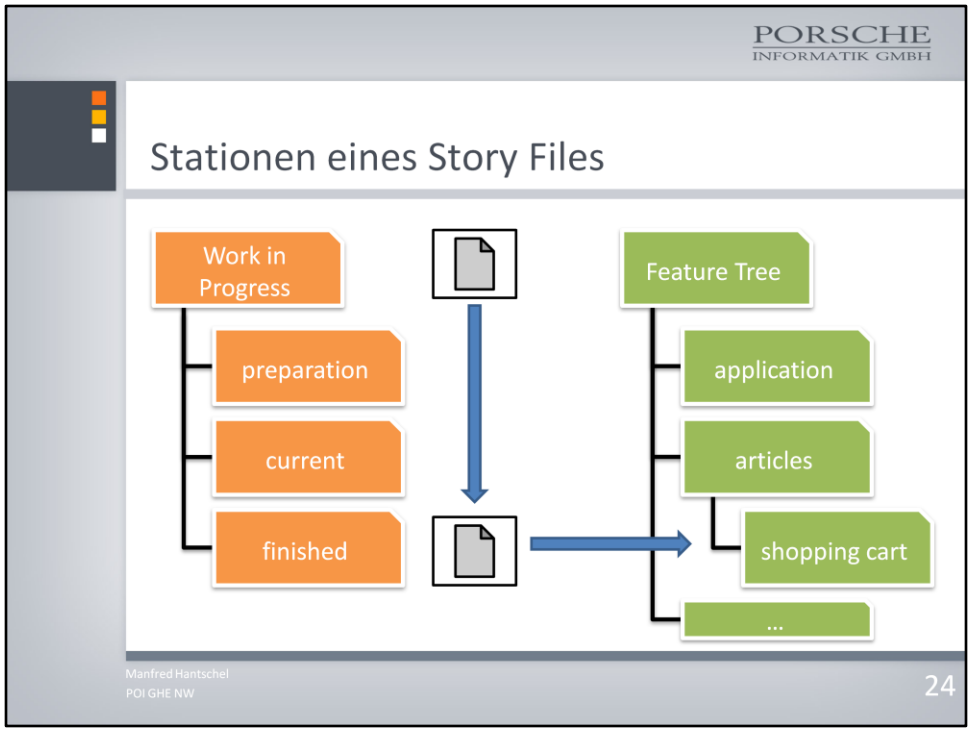


Kurz vor der Sprintplanung wird zwischen Entwickler und Tester abgestimmt, welche Verhaltensweisen man formalisieren und automatisieren will.

Entscheidungskriterien dabei sind:

- Wie wichtig ist das Behavior?
- Wie oft müsste es getestet werden?
- Wie aufwendig ist die Automatisierung?

Auf Basis dieser Entscheidungen werden dann die Szenarien formalisiert (was rausfällt landet in einem Testkatalog).

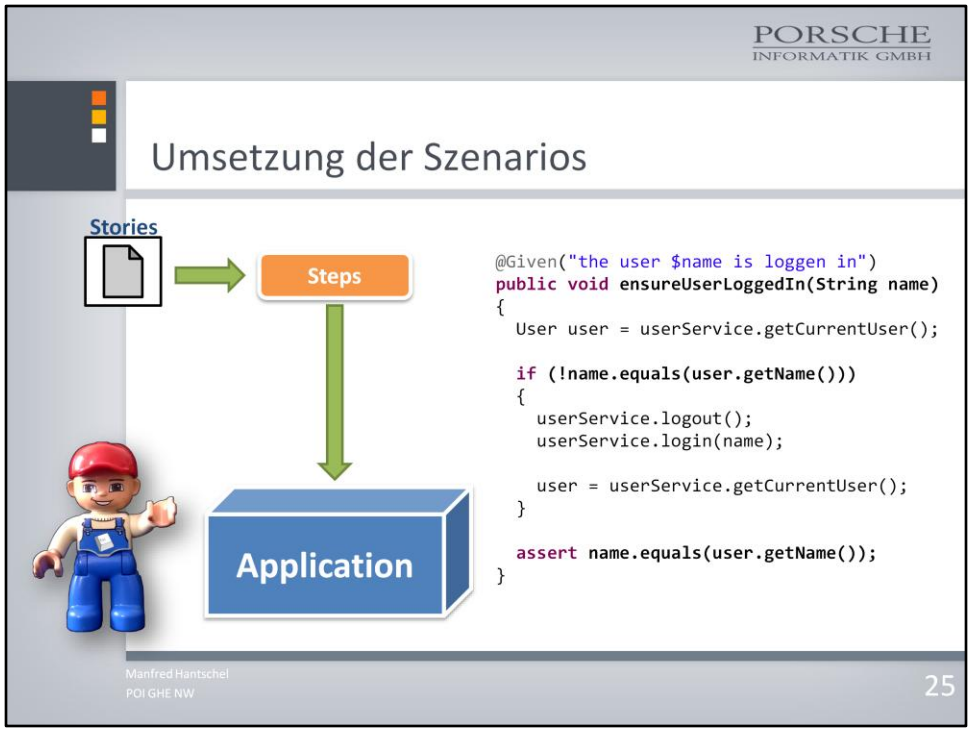


Im Entwicklungszyklus durchwandert ein Story File dann mehrere Stationen. Diese sind als Verzeichnisse im SVN abgebildet. Grundsätzlich wird zwischen Stories unterschieden, die sich in Entwicklung befinden und jenen, die fertig umgesetzt wurden und Teil der Systemspezifikation werden.

Zunächst befindet sich ein Story File in „preparation“. Soll die Story im Sprint umgesetzt werden, wird das File in „current“ verschoben. Wenn die Umsetzung erledigt wurde, wird das File in „finished“ verschoben.

Nach dem Sprint wird das Story File in den „Feature Tree“ eingepflegt. Der „Feature Tree“ ist nach den Modulen im System aufgebaut. Die Übernahme erfolgt entweder durch kopieren der Datei oder durch händisches Anpassen von bestehen Story Files.

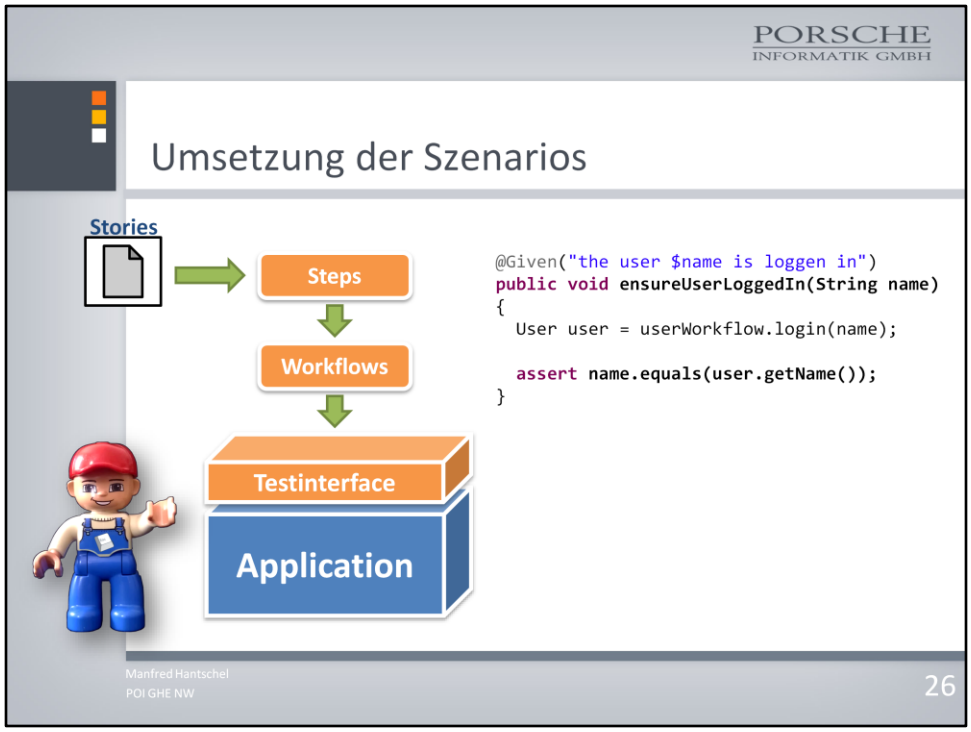
In einem Projekt der Porsche Informatik wird der „Feature Tree“ sogar dazu verwendet, automatisch eine Systemspezifikation zu generieren. Dazu werden die Story Files und zusätzliche Dateien im „Feature Tree“ zu einem Dokument zusammengefasst. Dieser Prozess ist leider sehr aufwendig, das Ergebnis ist jedoch beachtlich: Jede Version enthält eine getestete und exakte Systemspezifikation als PDF Dokument.



Wie erfolgt nun die Anbindung der Szenarios an die Applikation?

Grundlegend ganz einfach: in Steps-Klassen werden die Texte aus den Szenarien implementiert. Das Beispiel zeigt wie das grundsätzlich funktionieren könnte. Doch in dieser Art und Weise läuft man schnell Gefahr, dass die Wartungsaufwände für die BDD-Tests zu hoch werden.

Daher haben wir in diesem Bereich viel Energie investiert um die Tests robust zu gestalten.

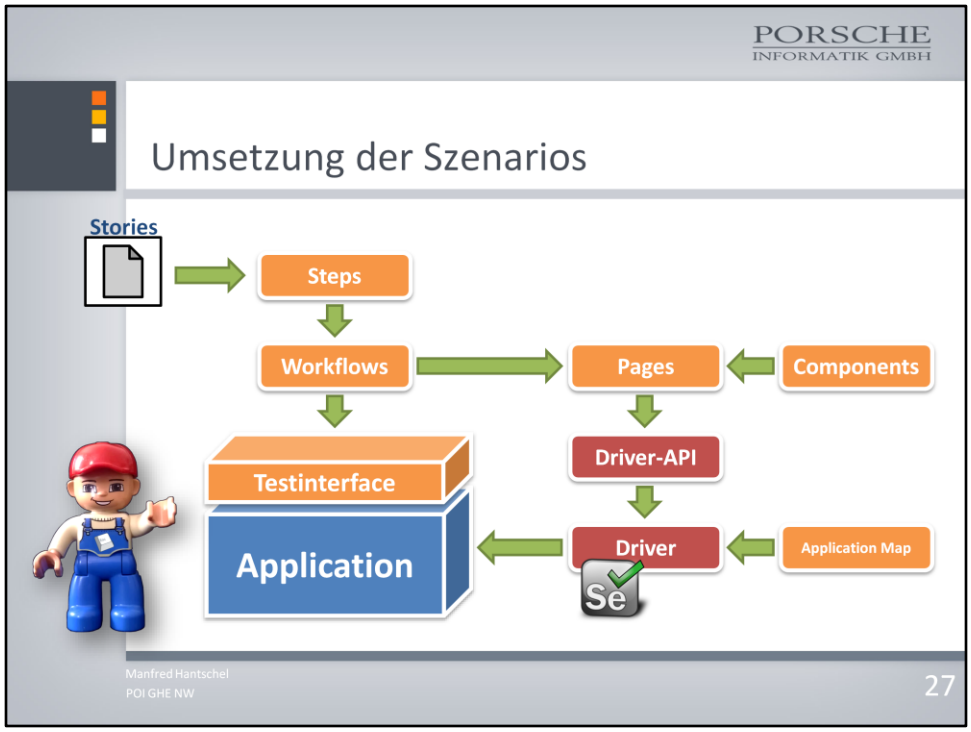


Grundsätzlich agieren die Steps nun auf Workflows. Erst die Workflows führen die Aktionen in der Applikation durch. Workflows agieren modulübergreifend, können daher auf die Schnittstellen der gesamten Applikation zugreifen. Die Logik wandert dadurch aus den Steps heraus und die BDD-Tests werden stabiler. Ähnliche Verhaltensweisen müssen auch nicht mehrfach implementiert werden.

Mit dieser Technik werden auch Testdaten aufgebaut. Die Tests müssen nicht auf einer vorbereiteten Datenbank laufen, die Tests erstellen die nötigen Testdaten selbst. Robustheit ist der Schlüssel.

Das Testinterface ermöglicht uns auch Aktionen auszuführen, die in der Applikation eigentlich nicht möglich wären, z.B. einen Benutzer zu löschen anstatt nur das Lösch-Flag zu setzen.

Bleibt aber immer noch das Problem die GUI zu steuern.

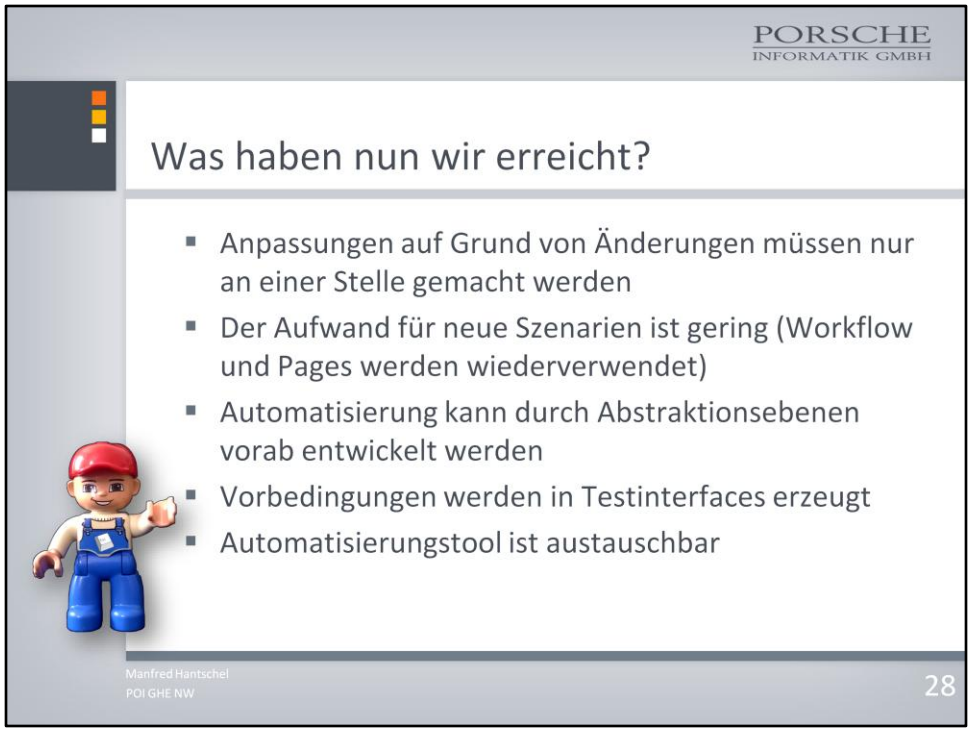


Um die GUI zu Steuern erzeugen wir für die Ansichten und Wartungen in der Applikation, Pages und Komponenten. Die Klassen beschreiben beispielsweise die Felder einer Login-Seite, also z.B. „Benutzername“, „Passwort“ und den „Anmelden“-Button. Zentrale Komponenten, wie z.B. einen Suchdialog, werden in Component-Klassen definiert, damit diese in mehreren Pages zur Verfügung stehen.

Die Pages interagieren über eine eigene Driver-API mit der GUI. In der API stehen die gängigsten Kommandos für die Automatisierung zur Verfügung, als z.B. „Klicke auf ein Element“, „Fülle Feld X mit Y“. Diese API soll uns ermöglichen auf Versionsupdates und Änderungen im Automatisierungstool zu reagieren.

Der Driver ist nun die spezielle Implementierung eines Automatisierungstools, in unserem Fall Selenium 2.x (Webdriver). In den Driver ist auch eine Application Map integriert. Sollte das Mapping von z.B. Feld-Name auf Feld-Id nicht automatisch möglich sein, kann dieser Definition ein Mapping hinterlegt werden.

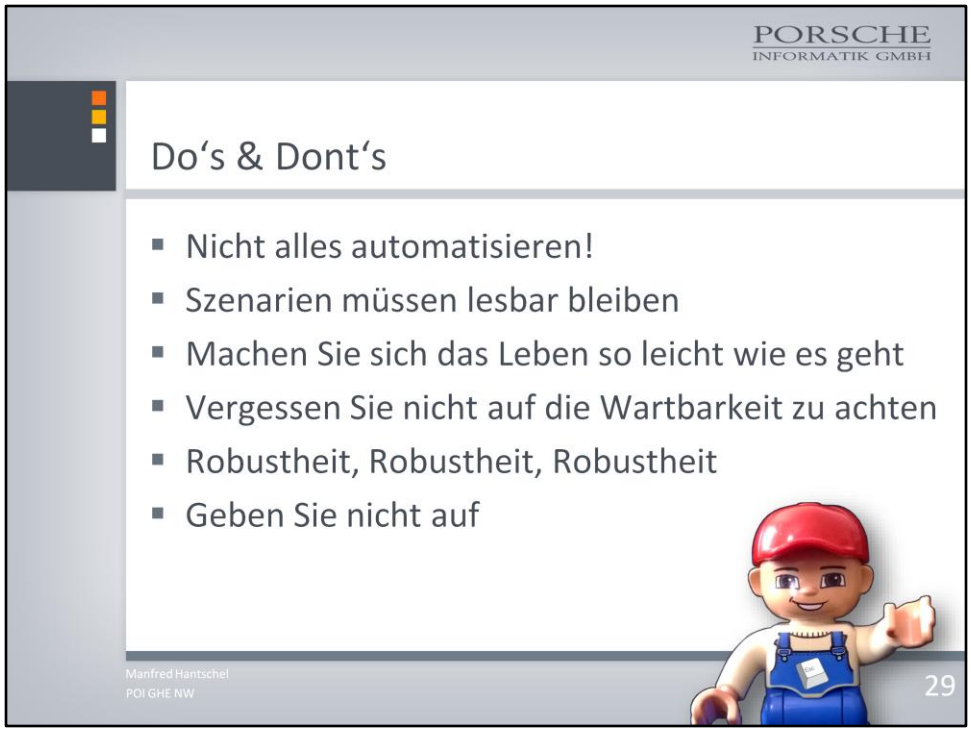
Die API und der Driver werden applikationsübergreifed eingesetzt.



Mit diesem scheinbar sehr komplexen Framework versuchen wir dem Wartungsproblem bei der Testautomatisierung
in den Griff zu bekommen. Wenn man das Framework nochmal genau betrachtet, stellt man schnell fest, dass der
Aufwand ein neues Scenario zu automatisieren darin besteht, die betroffenen Pages und deren Verbindung
untereinander zu beschreiben.

Zudem haben viele Scenarios dieselben oder zumindest sehr ähnliche Vorbedingungen – es stellt sich also sehr schnell
der Vorteil der Wiederverwendung ein.

Durch die Abstraktionsebenen und das automatische Mapping von abstrakten Feldbezeichnungen auf ID's ist es nun
auch problemlos möglich, GUI-Tests vorab zu automatisieren. Im Idealfall werden die Steps nun während der Entwicklung "grün".



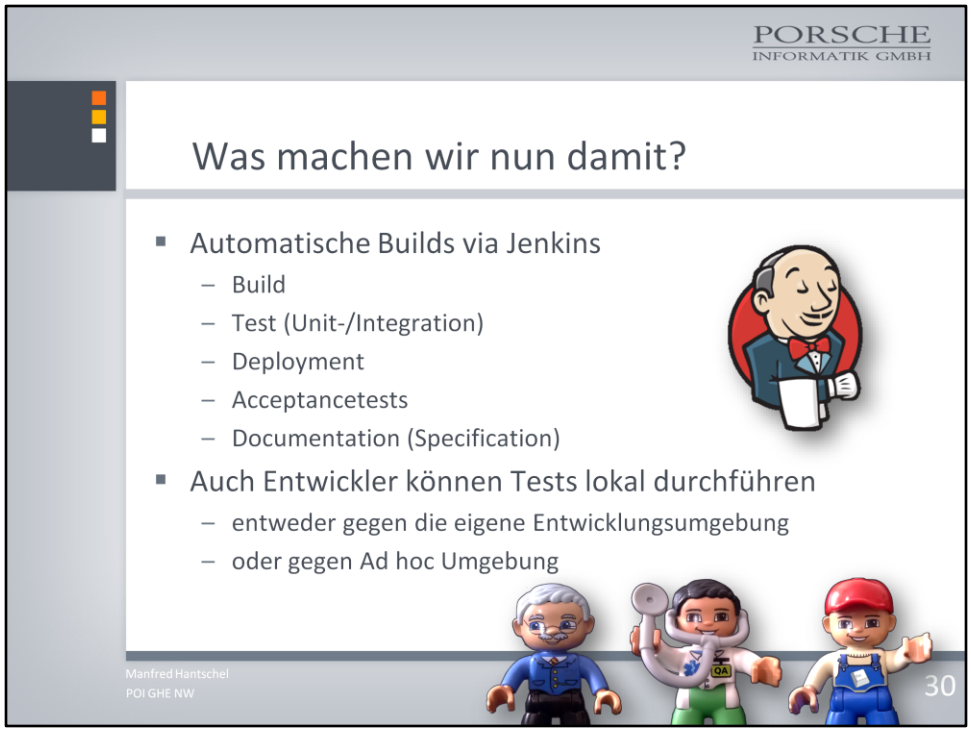
Nicht alles automatisieren
Je mehr man automatisiert, umso mehr muss man auch pflegen. Selbst wenn man ein ausgeklügeltes Framework verwendet, Anpassungen wird es immer geben.
Szenarien müssen lesbar bleiben
Verketten Sie nicht unzählige Vorbedingungen in einem Scenario nur weil es die Steps schon gibt. Verstecken Sie die Komplexität lieber in der Implementierung der Steps und formulieren Sie aussagekräftige Steps.

Machen Sie sich das Leben so leicht wie es geht
Je komplizierter der Weg vom Scenario bis zur fertigen Automatisierung ist, umso schwieriger wird es sein, ihre Entwickler zum Mitmachen zu bewegen.

Vergessen Sie nicht auf die Wartbarkeit zu achten
Steht etwas im Widerspruch zur obigen Aussage. Wenn man aber Steps mit Record&Replay Scripten automatisiert, ist man schnell in der Wartungsfalle und kommt nur unglaublich schwer wieder heraus.

Robustheit, Robustheit, Robustheit
Achten Sie bei Tests auf ihre Robustheit. Kein Test geht von Vorbedingungen aus, die nicht spezifiziert sind. Will ich einen Artikel verkaufen, muss einer verkaufbarer Arktikel verfügbar sein. Das „Given“ überprüft ob ein solcher Artikel vorhanden ist, legt wenn nötig einen an und merkt sich die Id, damit der Artikel im „When“ verkauft werden kann. Stories sollten ihre Testdaten aufräumen, und wenn die eigenen Testdaten zu Beginn schon in der Datenbank sind, trotzdem laufen.

Geben Sie nicht auf
Unsere Entwickler waren durch die Bank zumindest reserviert dem Vorhaben BDD gegenüber. Mittlerweile gibt es aber unzählige Befürworter des Vorgehens. Veränderung braucht Zeit und somit Ihre Geduld. Testen ist auch aufwendig. Wir haben das Glück, dass die Geschäftsleitung hinten uns steht und auf längere Sicht rentiert es sich.

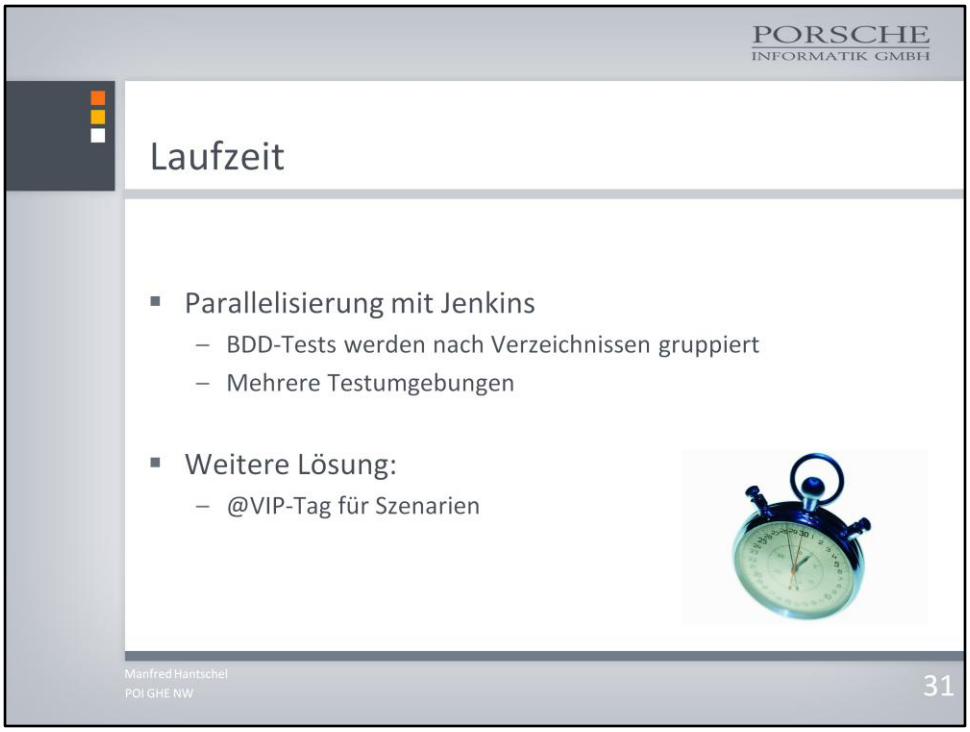


Wir verwenden Jenkins als CI-Umgebung. Dieser führ mehrmals am Tag einen vollständigen Build durch und deployed die Applikation automatisch auf ein Testsystem. Sobald der Applikationsserver gestartet worden ist, beginnt der Akzeptanztest. Bei Fehlern werden sowohl der Product-Owner als auch der zuständige technische Architekt per Email verständigt.

Anschließend wird noch die aktuelle Systemspezifikation generiert.

Dieses Vorgehen ist durch das Buch [Continous Delivery] inspiriert.

Natürlich können Entwickler Tests auch lokal durchführen. Dabei spielt die Robustheit der Tests wieder eine Rolle. Die Tests können selbst die Applikation starten, inkl. In-Memory-Datenbank, und die Tests dagegen ausführen.

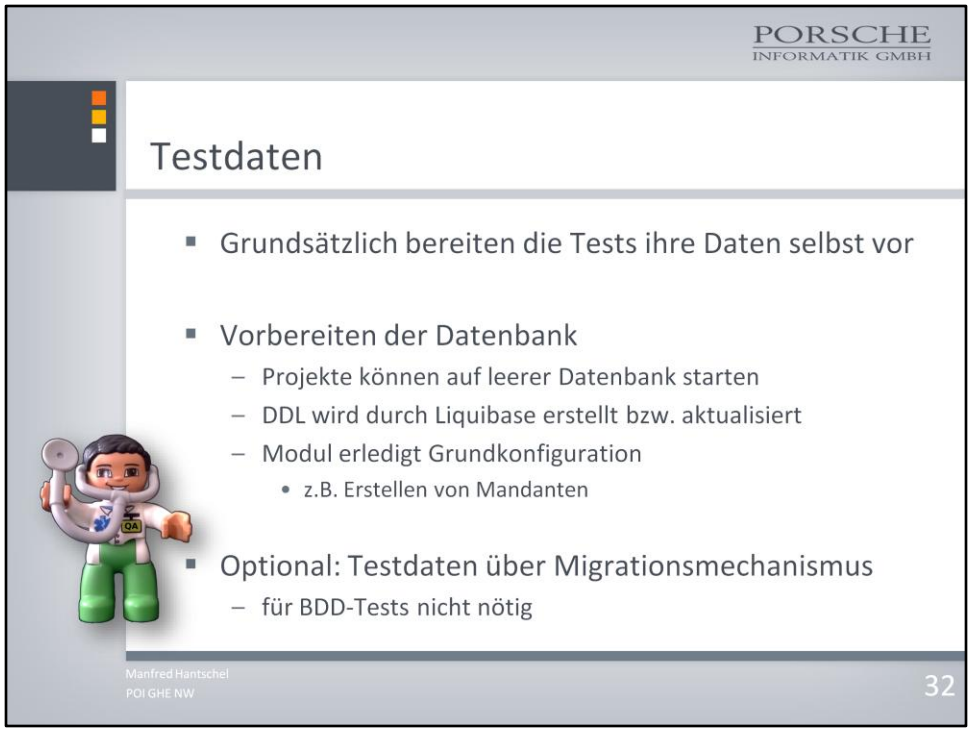


Sobald man mehrere Akzeptanztests hat kann die Laufzeit der Tests zu einem Problem werden. Jenkins kann Jobs parallelisieren und die Ergebnisse wieder vereinen. Die Akzeptanztests werden in mehrere Gruppen geteilt und parallel ausgeführt.

Zur Zeit haben wir noch in keinem Projekt große Probleme mit der Laufzeit. Die Parallelisierung haben wir daher erst bei einem Projekt umgesetzt. Da die Tests gegen Oracle laufen sollen, können wir keine echte Ad-hoc Umgebung verwenden, wir brauchen daher vier Testumgebungen, gegen die die Tests laufen (sonst könnte es vorkommen, dass sich die Test beeinflussen).

Auf Seiten der Testautomatisierungslösung verwenden wir dazu da Webdriver Grid2.

Sollte es wider Erwarten noch mehr Probleme mit der Laufzeit geben, haben wir für Szenarien von Tags vorbereitet. Im Buildzyklus kann dann definiert werden, dass nur Szenarien ausgeführt werden die mit @VIP gekennzeichnet sind.



Grundsätzlich ist es so, dass alle Tests ihre Daten selbst vorbereiten. Dies geschieht mit Hilfe der „Given“ Anweisungen. Hier ist wieder die Robustheit der Tests ein Thema.

Trotzdem ist es aber nötig, dass die Applikation beim Starten die Datenbank vorbereitet. Die Initialisierung bzw. die Aktualisierung erfolgt dabei durch Liquibase. Dann ist es noch notwendig eine Grundkonfiguration zu erstellen, also z.B. einen Mandanten für die Tests zu erzeugen.

Schließlich werden noch über einen Migrationsmechanismus, der in der Applikation sowieso existiert, Testdaten eingespielt. Diese Testdaten sind jedoch für die BDD-Tests nicht nötig. Sie dienen lediglich dazu die Umgebungen der Entwickler mit sinnvollen Daten zu befüllen.

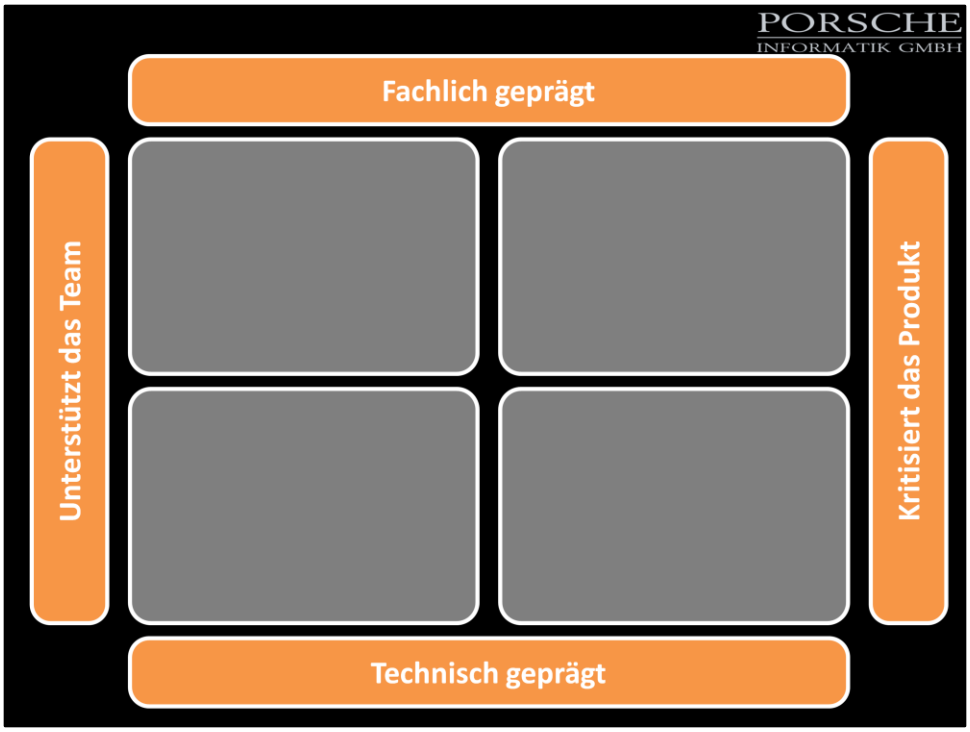
Beim BDD-Framework hat sich noch kein klarer Favorit herauskristallisiert. Zwei Projekte verwenden Cucumber in der Ruby-Implementierung und ein Projekt setzt auf jBehave. Beide Frameworks haben ihre Schwächen.

Zum Glück lassen sich die Frameworks leicht austauschen.

Cucumber ist eigentlich ein Ruby-Projekt. Im Build wird JRuby für die Ausführung verwendet und das macht mit Maven immer wieder riesen Probleme. Leider ist das ein ganz entscheidender negativer Punkt. Es gibt zwar mittlerweile Cucumber-JVM, aber das ist noch nicht fertig. Zumindest sind erste Release-Kandidaten verfügbar.

JBehave ist das gereiftere Projekt. Es hat jedoch Schwächen beim Reporting und bietet kein Tagging an. Außerdem ist der Story File Parser etwas instabil.

Prognose: Sobald Cucumber-JVM fertig ist, sollte es die BDD-Wars gewinnen.



Die [Testquadranten von Brian Marick] und in erweiterter Form von Lisa Crispin geben eine gute Orientierungshilfe in Bezug auf verschiedene Testarten.

Im Wesentlichen werden Tests nach 2 Ebenen kategorisiert:

Fachlich vs. Technisch geprägt

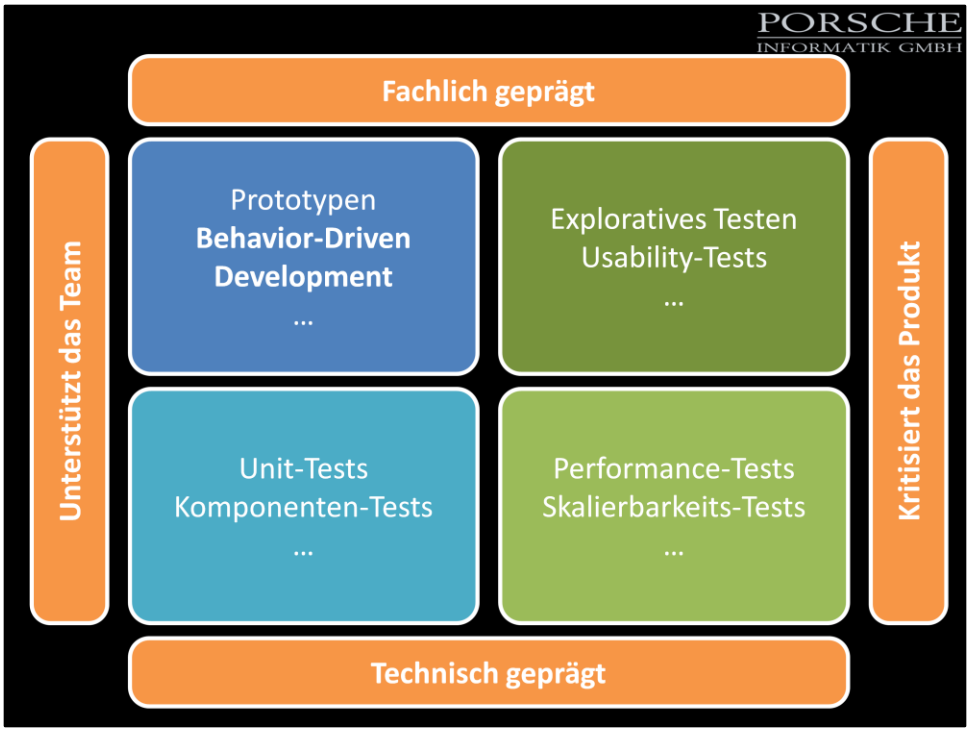
Die Unterscheidung zwischen technisch und fachlich geprägt kann man am leichtesten treffen, wenn man sich das Zielpublikum vor Augen hält mit dem man über einen Test spricht bzw. sich Gedanken macht, wer denn einen Test durchführen könnte. In einem technisch geprägten Test werden die Durchführenden wohl eher Entwickler oder zumindest Testautomatisierer sein wogegen bei fachlich geprägten Tests ein Tester oder vielleicht in Product-Owner bzw. sogar ein Kunde der Durchführende sein kann.

Unterstützt das Team / Kritisiert das Produkt

Man könnte auch sagen „vor der Entwicklung“ und „nach der Entwicklung“.

Entwickler die Erfahrungen mit Test-Driven-Development (TDD) gemacht haben, berichten oft, dass sie ihre Systeme/Klassen anders (einfacher) gestalten wenn die Tests vorher geschrieben werden – somit unterstützen diese Tests das Team während der Entwicklung. Außerdem werden durch die ständige Durchführung der Tests Seiteneffekte sofort entdeckt – eine weitere, nicht unwesentliche, Unterstützung.

Tester haben an und für sich einen destruktiven Job – sie versuchen das System zu brechen bzw. Fehler zu finden. Wenn ihnen das nicht gelingt, kann man mit hoher Wahrscheinlichkeit davon ausgehen, dass das System robust ist und die getesteten Funktionalitäten gewährleistet sind. Somit versuchen sie, das Produkt zu kritisieren und Schwachstellen aufzudecken.



Versucht man nun die verschiedensten Testarten in dieses System einzuordnen, fällt dies relativ leicht:

Exoplorative Tests, Usability Tests, Kundenabnahmetests

Diese Arten von Tests – meist manuell durchgeführt – versuchen Fehler im System zu finden, kritisieren also das Produkt. Auf der zweiten Ebene betrachtet sind diese Tests aber ausschließlich durch fachliche Anforderungen geprägt – man testet also gegen die fachliche Spezifikation.

Unittests, Komponententests, ...

Wenn man Unittests & Co nach TDD schreibt (und nicht nachgelagert um irgendwelche Kennzahlen zu befriedigen ☺), unterstützen sie das Team bei der Arbeit indem sie Einfluss auf die Systemarchitektur haben und Seiteneffekte sehr schnell erkannt werden. Allerdings sind sie technisch geprägt, werden also durch Entwickler entworfen und geschrieben.

Performancetests, Skalierbarkeitstests, ...

Auch diese Tests versuchen das Produkt zu kritisieren indem sie die nicht-funktionalen Anforderungen in Frage stellen. Sie sind technisch geprägt da ein Product-Owner wohl kaum in der Lage sein würde solche Tests zu entwickeln.

Prototypen, Behaviour-Driven-Development

Last but not least sind Vorgehensweisen wie Prototyping, Storyboards & Co sowohl fachlich geprägt als auch unterstützend für das Team. Eine (oft viel zu wenig beachtete) Grundregel in der Softwareentwicklung besagt: Je besser eine Anforderung vorbereitet und durchdacht ist, umso einfacher wird die Entwicklung. Hier ist auch das Behaviour-Driven-Development anzusiedeln!

ANHANG

Referenzen, Behavior-Driven Development

- „Introduction BDD“ von Dan North
  - http://dannorth.net/introducing-bdd/
- Behavior-Driven Webpräsents von Dan North
  - http://behaviour-driven.org/
- JBehave BDD Framework
  - http://jbehave.org/
- Cucumber Framework
  - JVM: https://github.com/cucumber/cucumber-jvm
  - Ruby (Cuke4Duke): http://cukes.info/

Referenzen, sonstige:

- Domain-Driven Development
  - http://www.domaindrivendesign.org/
  - Gleichnamiges Buch von Eric Evans, ISBN 978-0321125217
- Continuous Delivery
  - http://continuousdelivery.com
  - Gleichnamiges Buch von Jez Humble, David Farley, ISBN 978-0321601919
- Testquadranten von Brian Marick
  - http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1
- Liquibase, Database Change Management
  - http://www.liquibase.org/