A silhouette of a knight on a horse, holding a flag aloft, set against a dramatic, cloudy sky. The knight is positioned on the left side of the frame, facing right. The horse is rearing up slightly. The sky is filled with soft, white clouds against a darker blue background. The overall mood is one of historical grandeur and determination.

Auf dem **Kreuzzug** gegen
Fehler (in Software)

by Jerome Müller

A stage with red curtains and a wooden floor. The curtains are pulled back, revealing a dark stage floor. The text is centered on the stage.

Ein Stück in drei Akten

I - Der “Fehler-Entwicklungs-Prozess”

II - Agilität und Testing

III - Von der Idee zur Realität

Wo kommen die
Fehler her?



And THAT's the question...

Teil I:

Der “Fehler-Entwicklungs-Prozess”

Dinge die scheinbar funktionieren -
aber Fehler garantieren

Der erste Teil behandelt einige Dinge die auf den ersten Blick normal und richtig erscheinen - aber beim näheren hinschauen Fehler garantieren.

Problem 1:

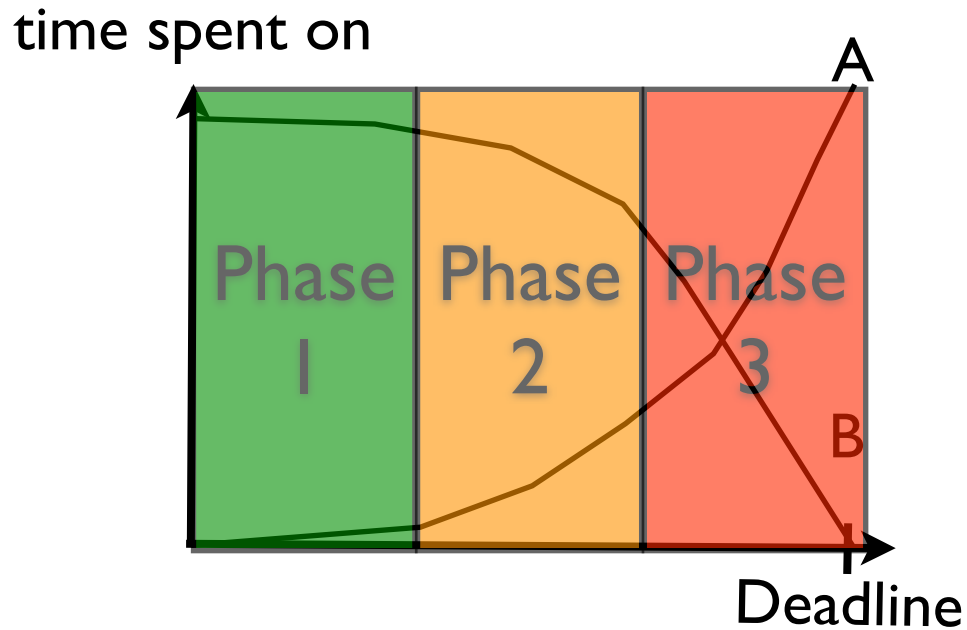
Bugzilla! Jira!

Als Kommunikationstool zwischen
Entwicklern und Testern

Ich will die Tools nicht generell als schlechtes Zeichen interpretieren. Aber wenn es als Kommunikationsmittel zwischen den Entwicklern und den Testern eingesetzt wird ist das völlig falsch! Wie viele Fehler sollen wir in die Software programmieren? Maximum 5? Wozu brauchen wir da ein Bugzilla? Wenn es natürlich nicht darauf ankommt wie viele Bugs wir erstellen und es wichtig ist so schnell wie möglich irgendwas zu machen - dann ist Bugzilla super. Rettet aber das Projekt auch nicht...

Bugzilla und Jira und wie sie alle heißen machen in gewissen Umfeldern sicher Sinn. Zum Beispiel für Open Source Projekte oder für die Kommunikation mit Kunden etc.

Problem 2:



A = Coding

B = Anforderungen verstehen

Hier ein verwandtes Problem: Die Y-Achse zeigt an wie viel Zeit wir für eine Arbeit aufwenden. Auf der X-Achse haben wir wieder die vergehende Zeit. Die erste Linie ist: Wie viel Zeit verbringen wir mit dem eigentlichen erstellen der Software (Coding). In der Phase 1 wird wenig gemacht - die Entwickler versuchen die Business Domäne zu verstehen, schauen sich Frameworks an etc. Je näher die Deadline rückt, desto mehr fokussieren sie sich auf die Programmierung.

Auf der anderen Seite kümmern sich die Entwickler am Anfang sehr stark um das korrekte Verständnis der Anforderungen. Je näher die Deadline kommt, desto weniger arbeiten sie daran.

Auf den ersten Blick scheint das Sinn zu machen. Wenn man die Anforderungen einmal verstanden hat, dann muss man es ja nur noch umsetzen.

Das trügt leider.

In Phase 1 verbringen die Entwickler viel Zeit bei Diskussionen mit den Leuten die die Anforderungen gestellt haben und darauf hin beim anpassen der Dokumente. In dieser Zeit werden praktisch keine Fehler erstellt - es wird ja auch nur wenig Code erstellt.

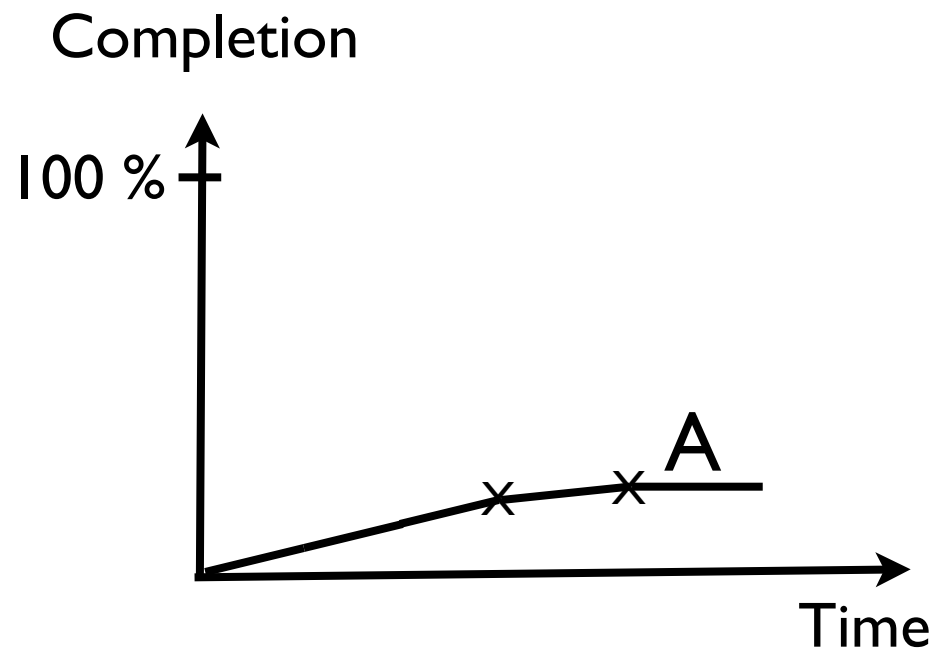
In Phase 2 beginnen die Entwickler mit der "richtigen" Arbeit - sie haben die Anforderungen einigermaßen Verstanden. Programmieren ist allerdings eine Detailarbeit - es kommen immer wieder Fälle vor, die nicht klar sind. Diese Fälle werden mit dem Kunden besprochen und meistens ins Anforderungsdokument aufgenommen.

In Phase 3 wird es langsam zeitkritisch. Die Deadline naht. Die Entwickler kennen die Domäne des Kunden schon recht gut. Sie haben jetzt keine Zeit mehr für Fragen (vor allem wenn der Kunde nicht grad "griffbereit" ist) und sie entscheiden z. T. selber wie sie die Anforderungen interpretieren. Wenn zwei Interpretationen etwa gleich wahrscheinlich macht es aus logischer Sicht Sinn diejenige zu wählen die weniger Arbeit gibt. Das Anforderungsdokument wird nicht mehr angepasst.

Das bedeutet dann zwangsläufig, dass das Anforderungsdokument nicht mehr mit der Software übereinstimmt. Es bedeutet auch, dass in der Testphase viele Fehler gefunden werden, die eigentlich gar keine sind.

Problem 3:

Bugfix “Iterationen”



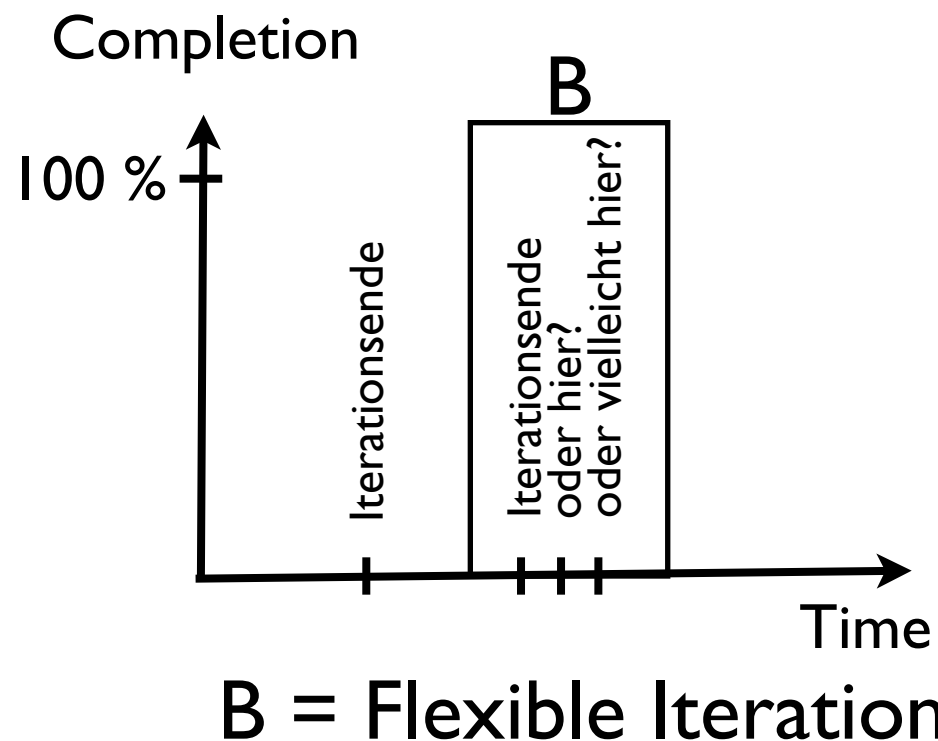
A = Bugfix Iteration

Die Bugfix Iteration kommt folgendermassen zu Stande: Die Entwickler programmieren eine Zeit friedlich vor sich her und erstellen mehr und mehr Features. Sie generieren auch Fehler, die nicht sofort entdeckt werden. Die versteckte Arbeit halt. Beim ersten “x” geht die Software zum ersten Mal an die Testabteilung über. Die Entwickler entwickeln weiter, kommen aber nicht mehr ganz so schnell voran, weil sie den Testern helfen müssen mit Infos wie: “Wie kann ich xyz testen?” “Wie setze ich die DB neu auf?” “Warum kann ich die Applikation jetzt nicht mehr starten?”

Am Ende dieser Iteration hat die Testabteilung 50 Fehler gefunden. Was passiert jetzt? Die Fehler müssen behoben werden - es wird eine Bugfix “Iteration” eingeschoben. Für die Projektleitung ist eine Bugfix Iteration eine Iteration die keinen Mehrwert generiert. Die Software kann nicht mehr als sie vorher schon hätte tun können.

Problem 4:

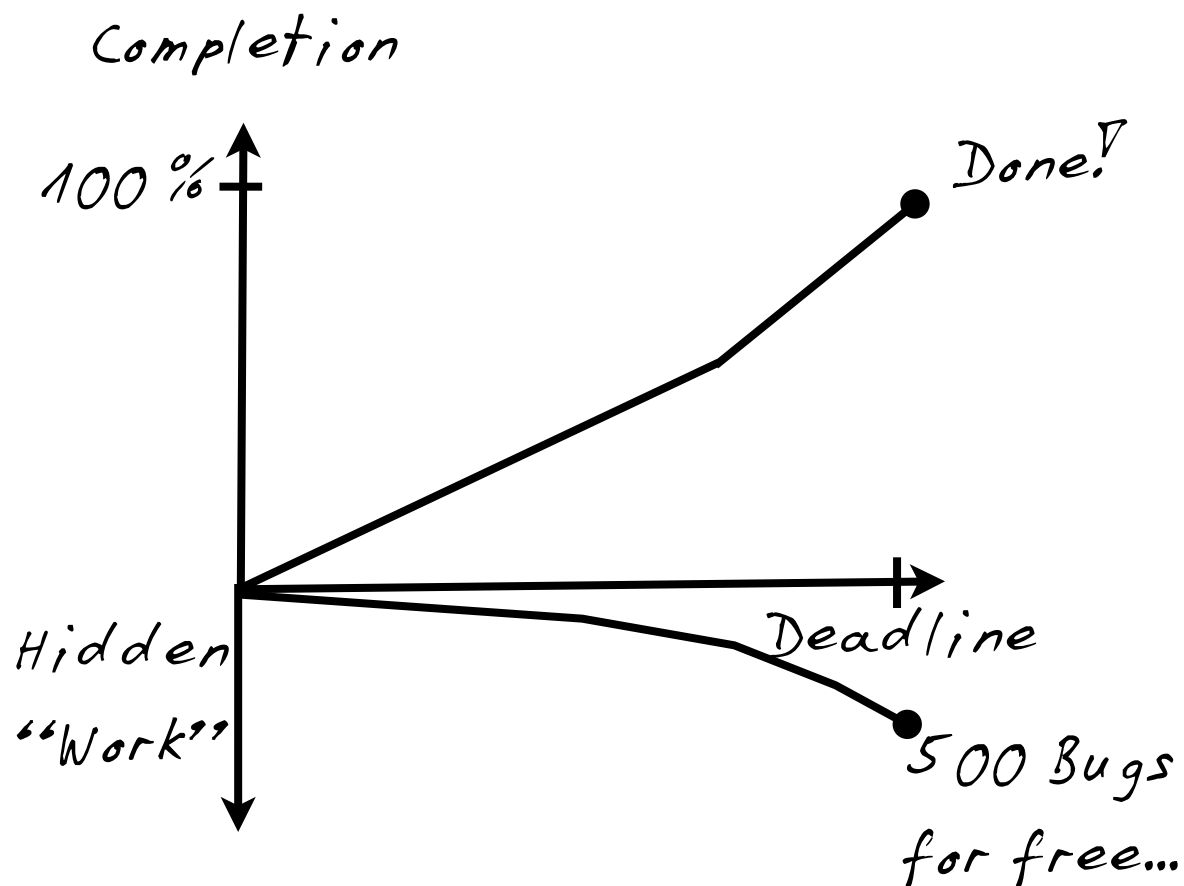
“Flexible” Iterationen



Die Worte Flexibel und Iteration sollten nie nacheinander kommen. Iterationen ist ein fixer Zeitraum. Der wird nicht ausgedehnt. Wenn man sagt 2 Wochen, dann sind es nicht plötzlich 3 oder 4. Das ist einer der wichtigen Punkte der agilen Entwicklungsmethoden. Nach 2 Wochen ist man fertig. Eventuell hat man NICHT ALLES fertig. Aber man kann den Fortschritt (oder die Absenz davon) zeigen und festlegen wies weiter gehen soll.

Nach 1.8 Wochen feststellen, dass man noch viele Fehler hat und nicht fertig wird und deshalb die Iteration um 1 Woche ausdehnt - das ist vieles - aber ganz sicher kein agiles Vorgehen.

Problem 5: Fertig?



Auf der Y-Koordinate haben wir den Prozentgrad der Fertigstellung der Software. Auf der X-Achse ist die Zeit die vergeht und die Deadline, die gesetzt ist. Die Entwickler machen stetigen Fortschritt und merken dann nach etwa zwei Drittel dass es knapp wird. An diesem Punkt beginnen sie härter, schlauer und mehr zu arbeiten und werden genau auf die Deadline fertig. "YEAH!" - Zeit für die Entwickler bei einem Aperero anzustossen und sich gegenseitig auf die Schulter zu klopfen.

Was allerdings passiert ist, dass neben der eigentlichen Arbeit noch zusätzliche Arbeit gemacht wird. Versteckte Arbeit. Oder Negative Arbeit. Je näher wir zur Deadline kommen, desto mehr unentdeckte Fehler sind in der Software.

Fertig heisst für die Entwickler eigentlich nur: "Wir haben alle Features implementiert und jedes Features hat mindestens einmal Fehlerfrei (zumindest fast) funktioniert."

Problem 6: ^{Datenbank}unabhängigkeit

Konfigurierbar

Zu testende Software ist für alles designed!

Reliabilität

Compliance

Services

Extensibility

Scalability

Plattformunabhängigkeit

Maintainability

...aber nicht um getestet zu werden...

Nehmen sie sich einmal die Mühe und fragen sie einen Entwickler wie die Applikation aufgebaut ist. Mit grösster Wahrscheinlichkeit wird er Aussagen machen wie: "Der Business Layer benutzt dann den Persistence Layer. Den Persistence Layer haben wir um die Datenbankunabhängigkeit zu gewährleisten. Und den Business Layer können wir auf verschiedenen Maschinen deployen und so skalieren."

Die Software wird für alles designed! Wechsel von Oracle auf XML Files? Kein Problem! Wechsel von Windows auf Linux? Kein Problem! Anstelle der Logfiles sollen die Logeinträge an einen Webservice geschickt werden? Piece of cake!

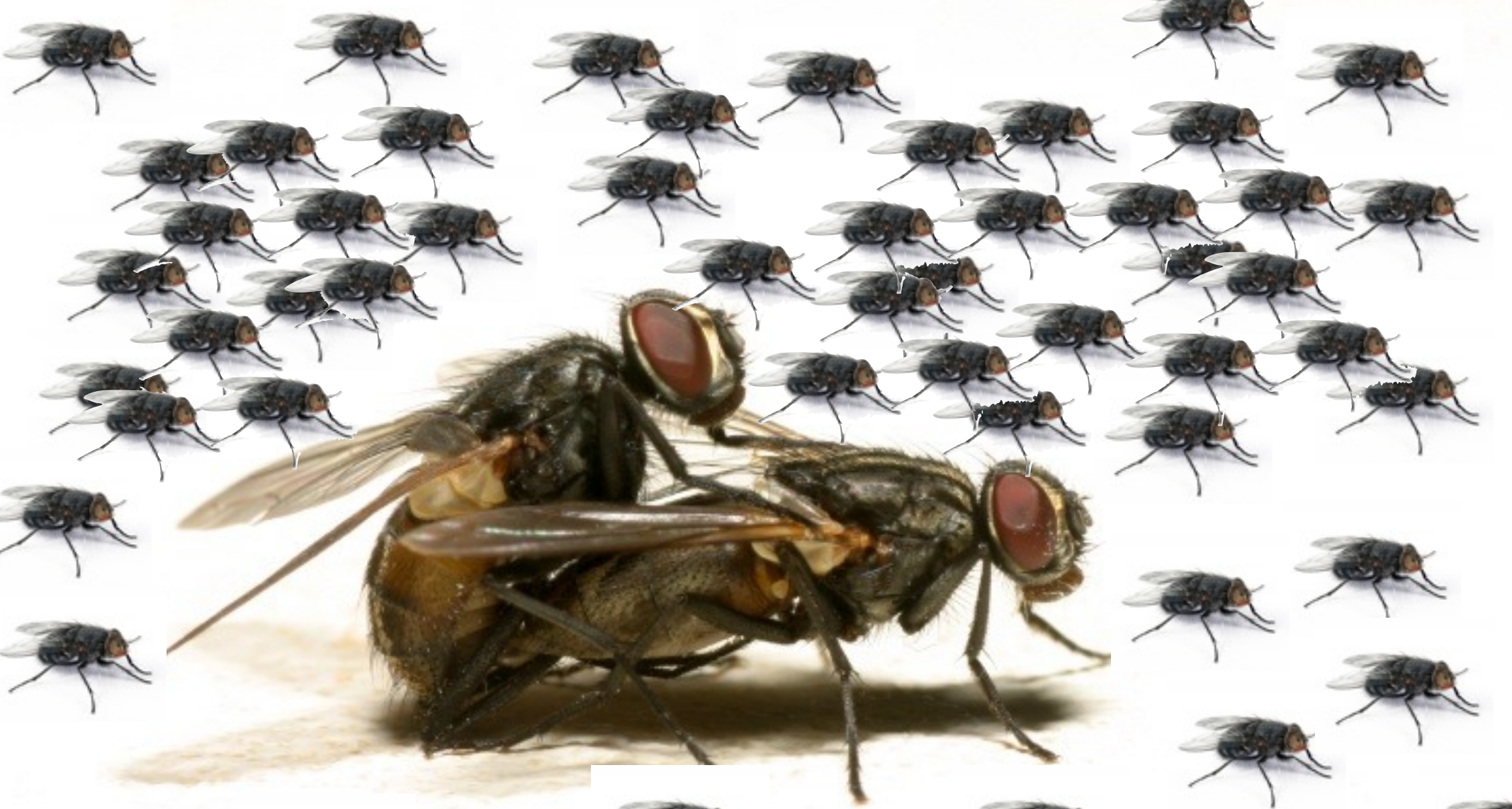
Nachdem sie dann den Entwickler entsprechend gelobt haben, fragen sie ihn einmal welche Konzepte er eingebaut hat, dass die Applikation "testbar" wird. Kann ich jede Bildschirm Maske innerhalb von 3s öffnen und mit einstellen, in welchem Zustand die angezeigten Daten sind, damit ich das UI Verhalten testen kann? Wie erstelle ich Testdaten? etc.

Die Antwort ist leider meistens sehr kurz und kann in Null Worten zusammengefasst werden.

Mit anderen Worten: Wir Entwickler designen die Software für alle Eventualitäten - aber wir designen sie nicht, damit sie einfach zu testen ist. Sorry - aber wir dachten das testen ist nicht unser Problem...

Diese Verhalten
sind...





...Brutstätten für Bugs

Diese Verhalten - obwohl sie auf den ersten Blick sinnvoll erscheinen - gewährleisten dass die Software voller Fehler ist. Die Frage ist, können wir diese Verhalten verändern. Und zwar so dass die Wahrscheinlichkeit Fehler einzubauen minimiert wird?

Wie können wir diese
Probleme
verhindern?



And THAT's the question...



Teil II:

Agilität und Testing

Auf der Jagd nach Bugs

Und die agilen Software Entwicklungsmethoden bringen eine Antwort. Diese Methoden definieren den Ablauf zum erstellen von Software neu (im Vergleich zu Wasserfall) - und dadurch auch die Verhaltensweisen.

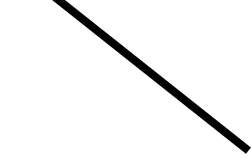
Einer der fundamentalen Grundsätze von Agilen Entwicklungsmethoden sind die Iterationen. Ein Konzept das zwar einfach ist, aber praktisch immer falsch angewendet wird. Es ist auch ein Konzept, das es ermöglicht den Bugs das Leben schwer zu machen.

Agiles Einmal Eins: Iterationen

1 - Timeboxed

2 - Endet mit FERTIGER Software

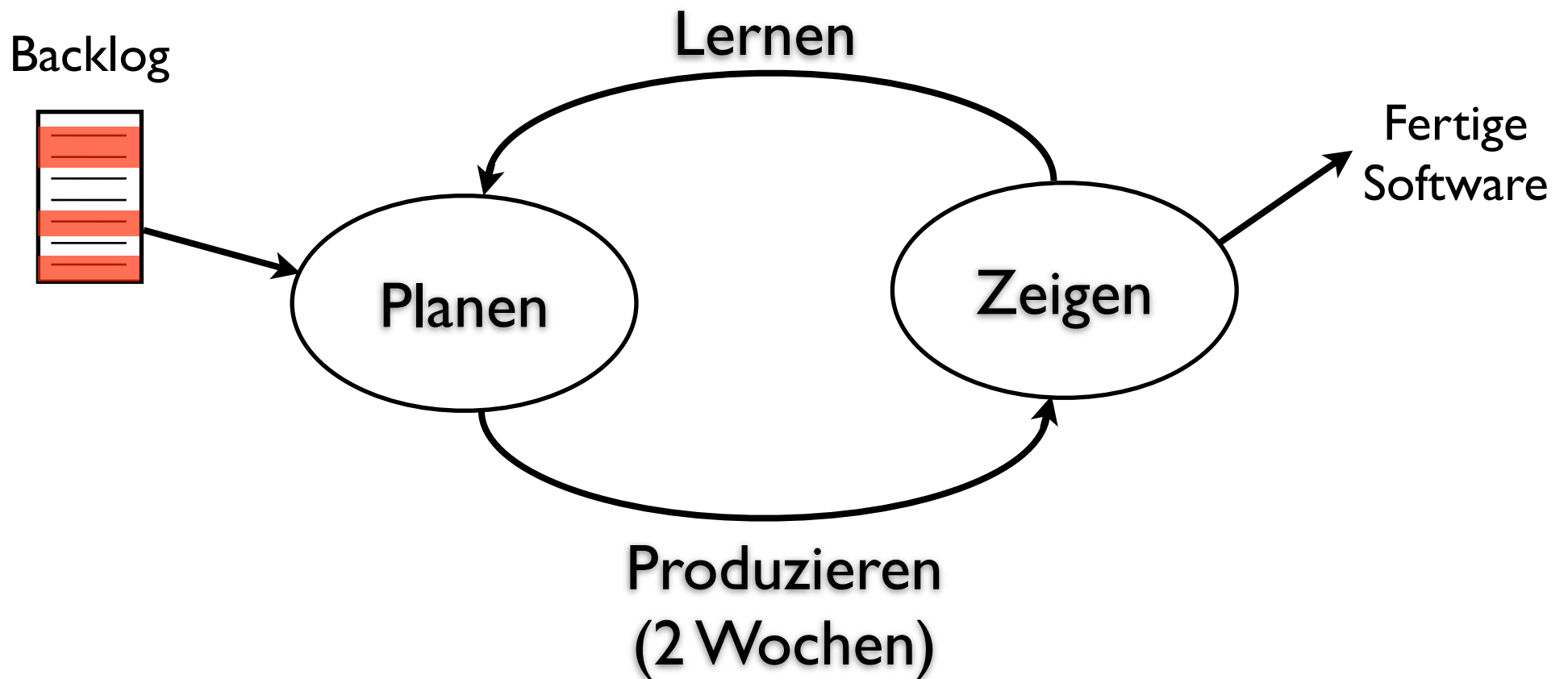
Impliziert dass die Software getestet wurde!



Eigentlich gibt es nur zwei Punkte zur Definition einer Iteration benötigt werden:

1. Timeboxed. Iterationen haben eine bestimmte Länge (z. B. 2 Wochen). Nach 2 Wochen ist die Iteration abgeschlossen. Es gibt keine "Verlängerung". Die Iterationen geben dem Projekt einen Rythmus - oder Puls. Wenn der Puls einmal ausbleibt, ist das schon ein Grund zur Sorge.
2. Software ist fertig. Das heisst installiert und funktionsfähig. Ein Architekturdokument gilt nicht als lauffähige installierte Software. Test Dokumente auch nicht. Und ein Feature das Fehler hat auch nicht. Fertige Software bedeutet, dass wir am Ende einer Iteration immer feststellen können, wie gut wir vorwärts kommen. Dadurch wird das Projekt steuerbar, weil der Projektfortschritt sichtbar wird.

Scrum



Hier ist die Entwicklungsmethode "Scrum". Ein bisschen abstrahiert. Auf dem Backlog befinden sich Features die implementiert werden müssen. Der Scrum beginnt mit Planung. Es wird entschlossen, welche Features von der Liste genommen werden und in den nächsten 2 Wochen implementiert werden. Danach werden diese Features produziert. In dieser Zeit kommen keine "zusätzlichen" Features dazu. Oder andere, weil sich die Prioritäten geändert haben. Das Team ist dafür verantwortlich, dass die Features (und alle vorherigen) am Schluss der Iteration fehlerfrei laufen.

Denn dann wird die Software demonstriert - das heisst sie ist installiert und sie funktioniert (hoffentlich - sonst sieht das Team ein bisschen alt aus an der Demo). Nach der Demo nimmt man sich die Zeit und überlegt: Gibt es Dinge, die wir besser machen können? Gibt es Probleme die auf uns zukommen, wenn wir so weitermachen?

Danach nimmt man die nächsten Features vom Backlog und erweitert die Software.

Aber wann kommt
die Testphase?

Scrum:

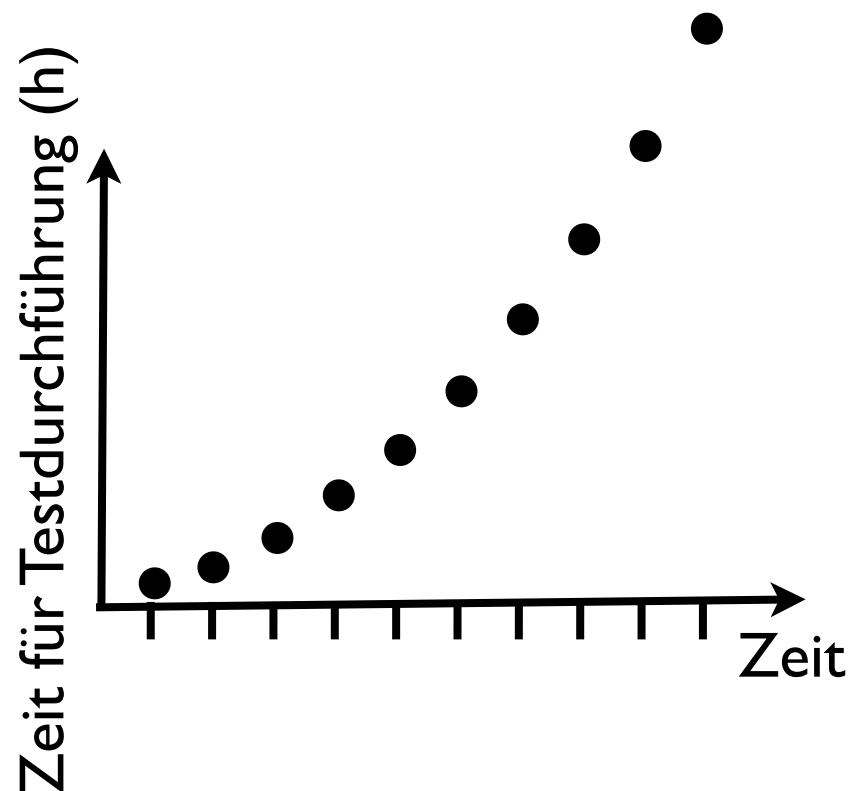
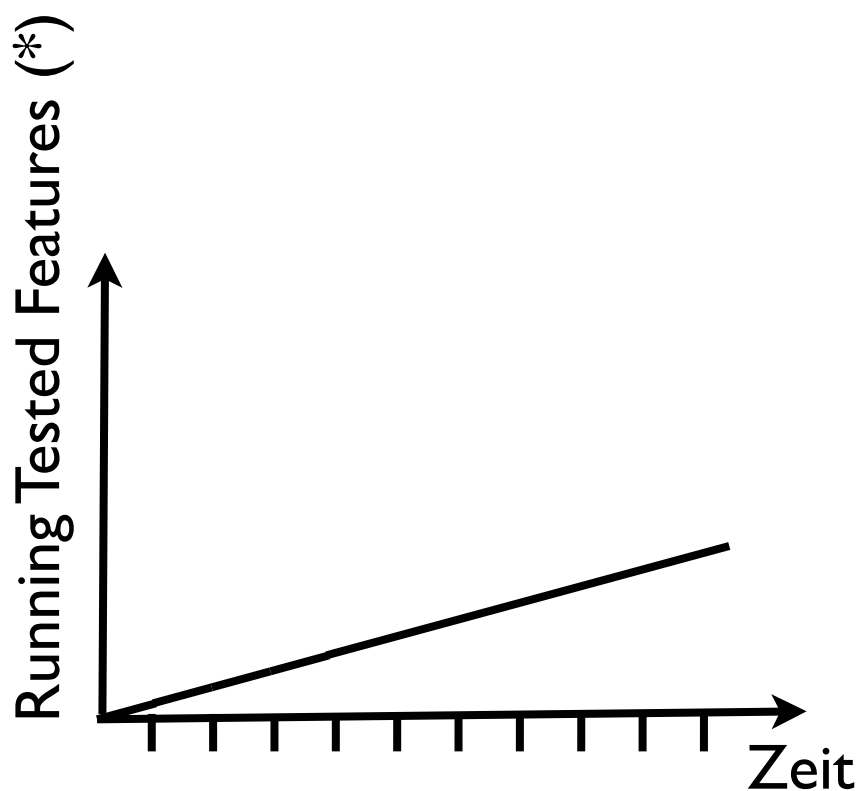
Kurzbeschreibung von Ron Jeffries

1. Produce Done-Done software on a regular basis (*)
2. Remove every obstacle to doing item 1.

(*) 2 - 4 Wochen.

Done-Done Software heisst, dass die Software getestet und fehlerfrei ist. OK, nun die Frage - hilft eine separate QA-Abteilung regelmässig alle 2 Wochen fertige Software zu liefern? Nein - dann siehe Punkt 2!

Und wann wird getestet?



(*) Vorgeschlagene Metrik von Ron Jeffries (siehe: <http://www.xprogramming.com/xpmag/jatRtsMetric.htm>)

Auf der ersten Y-Achse ist die Metrik "Running Tested Features". Auf dieser Achse geht man nach oben, wenn 1 Feature implementiert ist UND all seine Tests erfolgreich besteht. Auf der zweiten Y-Achse ist die Zeit, die für die MANUELLE Testdurchführung gebraucht wird.

Nach 2 Wochen ist die erste Iteration abgeschlossen. Es sind 2 Features implementiert worden, die problemlos am Ende der Iteration manuell getestet werden können. Der Testaufwand ist vielleicht 1 h. Nach der zweiten Iteration sind 2 weitere Features dazu gekommen, die z. T. das Verhalten der ersten etwas beeinflusst. Um die Software zu testen, müssen alle Tests der ersten Iteration durchgeführt werden plus alle neuen. Der Aufwand ist immer noch absehbar, vielleicht 2.5 h.

Aber während die Entwicklung Iteration für Iteration mehr Features produziert, brauchen die Manuellen Tests immer länger (wir müssen ja immer wieder die ganze Vergangenheit testen). Nach einigen Iterationen brauchen die Manuellen Tests 10 h. Das bedeutet, dass der Code jetzt schon 2 Tage vor Ende der Iteration fertig sein muss. Einige Iterationen später dauern die manuellen Tests 1 Woche. Das heisst die Entwickler haben noch 4 Tage zum entwickeln. Irgendwann brauchen die Manuellen Tests länger als die Dauer der Iteration... Was machen wir dann?

Automatisierte Tests
müssen Bestandteil der
Entwicklungsmethode
werden!!!

Mit anderen Worten: Automatisierte Tests müssen Bestandteil der Entwicklungsmethode werden. Wenn wir das nicht tun, dann müssen wir einen der beiden anderen Wege begehen - und die funktionieren nicht. Wie binden wir also die zu automatisierenden Tests in den Prozess ein?

Requirements.doc

Wozu?

Planen + Lehrmittel + Beweisen

Einfach gesagt: als Spezifikation bevor das Codieren beginnt. Aber ich möchte da noch etwas ausholen um zu zeigen warum das Sinn macht.

In einigen Projekten gibts ein Requirements.doc. Wozu benutzen wird das während der Entwicklung? Eigentlich für 3 Sachen. Wir brauchens um Schätzungen zu machen, um zu lehren wie das Business funktioniert und um zu beweisen, dass die erstellte Software das tut was sie soll.

Requirements the Agile Way

Planen = Feature Beschreibung

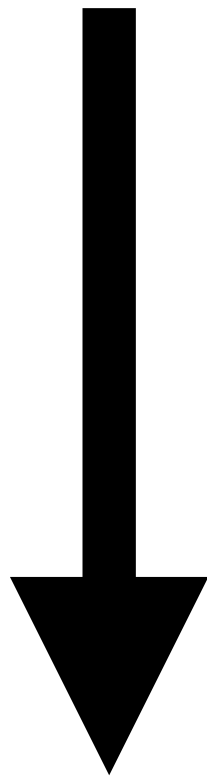
Lehren = Konversation

Beweisen = Automatisierte Tests

Die meisten agilen Methoden brauchen für diese drei verschiedenen Aspekte auch drei verschiedene Mittel. Für die Planung reichen Feature Beschreibungen. Das Erlernen der Business Domäne passiert vorzugsweise über Konversationen und das Beweisen ist am besten durch automatisierte Tests gelöst.

Das bedeutet aber auch, dass die automatisierten Tests nicht kryptisch sein dürfen. Das heisst, ich muss sie meiner Grossmutter geben können und sie muss sie verstehen. Und das ist da, wo ein Word Dokument brilliert. Da es "normale" Worte sind wie sie in jedem Buch zu finden sind, sollte sie auch jeder verstehen...

Zeit für ein kleines Praxis Beispiel



Feature
Konversation
Test(s)
Code

Hier nun der Ablauf, wie wir vom Feature zum Code kommen.

Das Feature haben wir ja bereits ausgewählt. In der Konversation versuchen wir Details zum Feature rauszufinden und schreiben nachher Tests, die unser Verständnis des Features ausdrücken (wie gesagt, der Test muss für jeden verständlich sein). Danach wird der Code geschrieben, damit diese Tests erfolgreich durchlaufen.

Eventuell tauchen bei der Implementation Fragen auf – worauf die Konversation wieder aufgenommen wird, die Frage geklärt und neue Tests geschrieben werden. Der Code wird angepasst oder erweitert. So wachsen unsere Requirements, die automatisierten Tests und der Code gleichzeitig. Die Software wird immer vollständiger.

“Fertig”

1 Feature => X Tests => Alle Grün

Die Tests die wir gesehen haben sind für ein Programm sehr einfach zu parsen und auf die entsprechende Business Logik zu mappen. Die Tests sind aber auch für jeden mit nur wenig Aufwand verständlich.

Fertig im agilen Sinne heisst also: Wir haben ein Feature, das durch X Tests definiert wird. Wenn ALLE Tests erfolgreich durchlaufen, ist das Feature vollständig implementiert.

Das ist nur
der **erste Schritt.**

Wie gehts **weiter?**



Hmmm. Das tönt ja alle recht interessant und spannend, aber warum ist das wichtig. Oder vielmehr: was bedeutet das für die Testabteilung.

Naja, wenn wir die automatisierten Tests während der Entwicklung machen und wir die Testphase abschaffen. Was passiert dann mit den Testern?



Teil III: Von der Idee zur Realität

Und die agilen Software Entwicklungsmethoden bringen eine Antwort. Diese Methoden definieren den Ablauf zum erstellen von Software neu (im Vergleich zu Wasserfall) - und dadurch auch die Verhaltensweisen.

Einer der fundamentalen Grundsätze von Agilen Entwicklungsmethoden sind die Iterationen. Ein Konzept das zwar einfach ist, aber praktisch immer falsch angewendet wird. Es ist auch ein Konzept, das es ermöglicht den Bugs das Leben schwer zu machen.

Idee zur Realität



In welcher Zeiteinheit
wird x in ihrem Projekt gemessen?

“Demo! Heute!”

Was funktioniert?

Wie lang bis Demo?

Bug Risiko?



Unit Tests
Check In Regeln

Automatische Tests
Kleine Features

Automatisches
Deployment

Code

Testing

Deploy

Integration

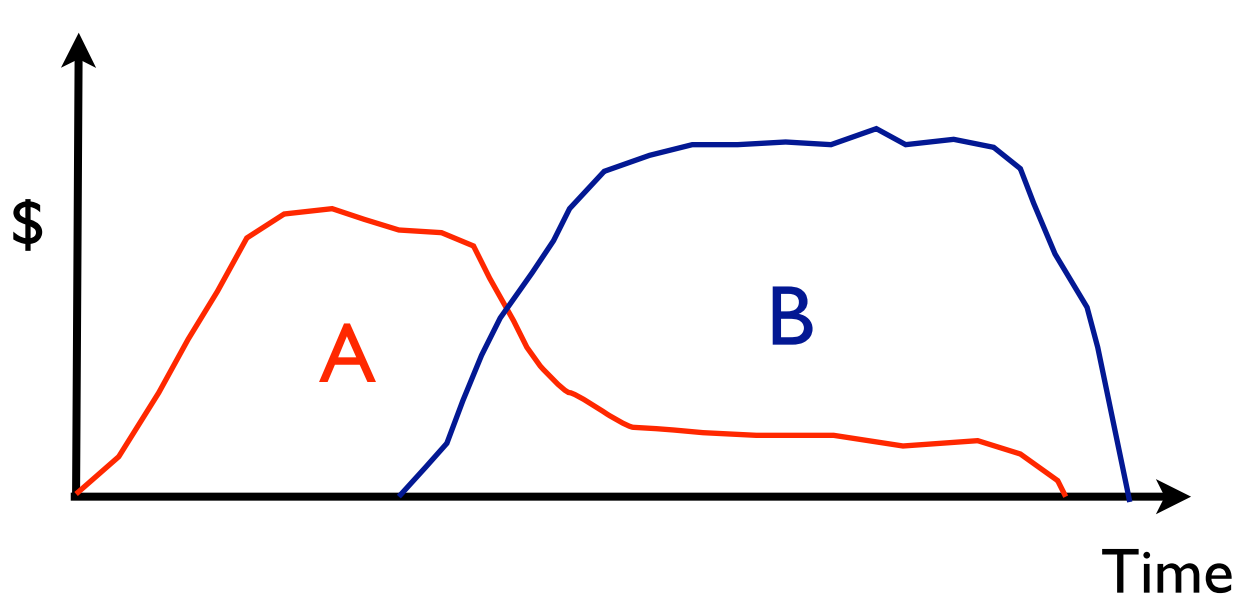
Daten Migration

Automatischer Build
Continuous Integration

Diff Scripts
Rollback Scripts

```
graph TD; UT[Unit Tests] --- C[Code]; CIR[Check In Regeln] --- C; AT[Automatische Tests] --- T[Testing]; KF[Kleine Features] --- T; AD[Automatisches Deployment] --- D[Deploy]; C --- H[ ]; T --- H; D --- H; H --- I[Integration]; H --- DM[Daten Migration]; I --- AB[Automatischer Build]; CI[Continuous Integration] --- AB; DM --- DS[Diff Scripts]; RS[Rollback Scripts] --- DS;
```

Warum wir
bezahlt werden



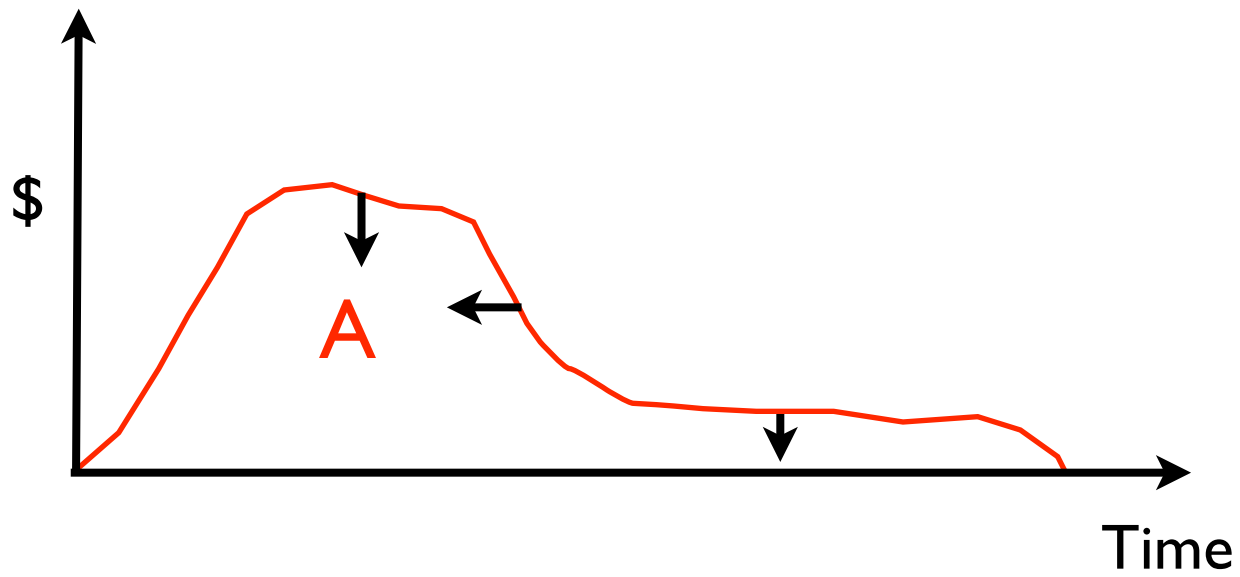
A = Kosten

B = Nutzen

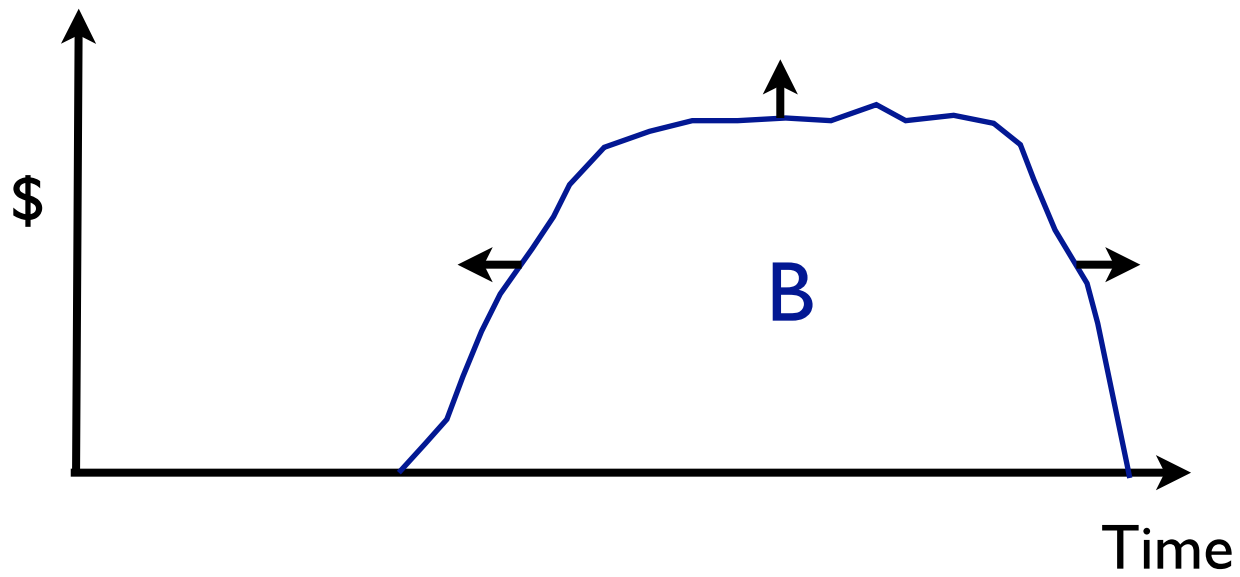
$B > A$

Billiger machen

A = Kosten



Mehr Nutzen



B = Nutzen

Testphase?

Code

Testing

Deploy

Integration


Daten Migration

blau = Aufgabe Entwicklungsteam

rot = Aufgabe vom Testteam



Angenommen es ist
wahr, dass...

- ...der Prozess falsch ist.
 - ...wir Bugs verhindern können.
 - ...es keine Testphase braucht.
- 

Wie verändert sich dadurch der Job der Tester?



Hmmm. Das tönt ja alle recht interessant und spannend, aber warum ist das wichtig. Oder vielmehr: was bedeutet das für die Testabteilung.

Naja, wenn wir die automatisierten Tests während der Entwicklung machen und wir die Testphase abschaffen. Was passiert dann mit den Testern?