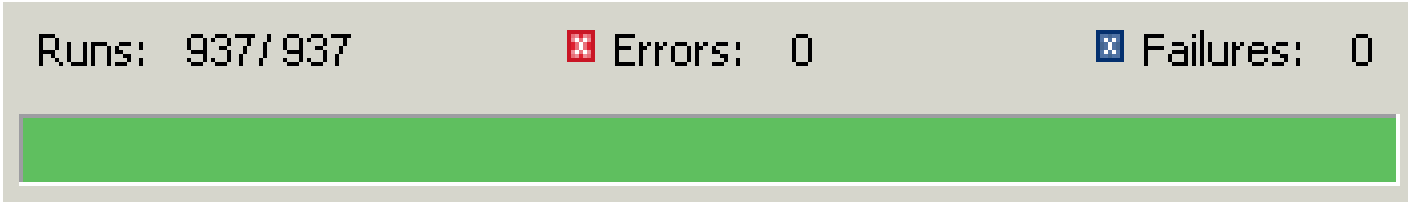
OBJECT VISION
SOFTWARE

Automatisierte Entwicklertests



Erfahrungen und Ergebnisse einer Prozessanpassung

Michael Borgwardt - 29.04.2008

Ablauf

1. Voraussetzungen / Umfeld
2. Umsetzung / Hürden
3. Ergebnisse / Lessons learned

Ablauf

1. Voraussetzungen / Umfeld
2. Umsetzung / Hürden
3. Ergebnisse / Lessons Learned

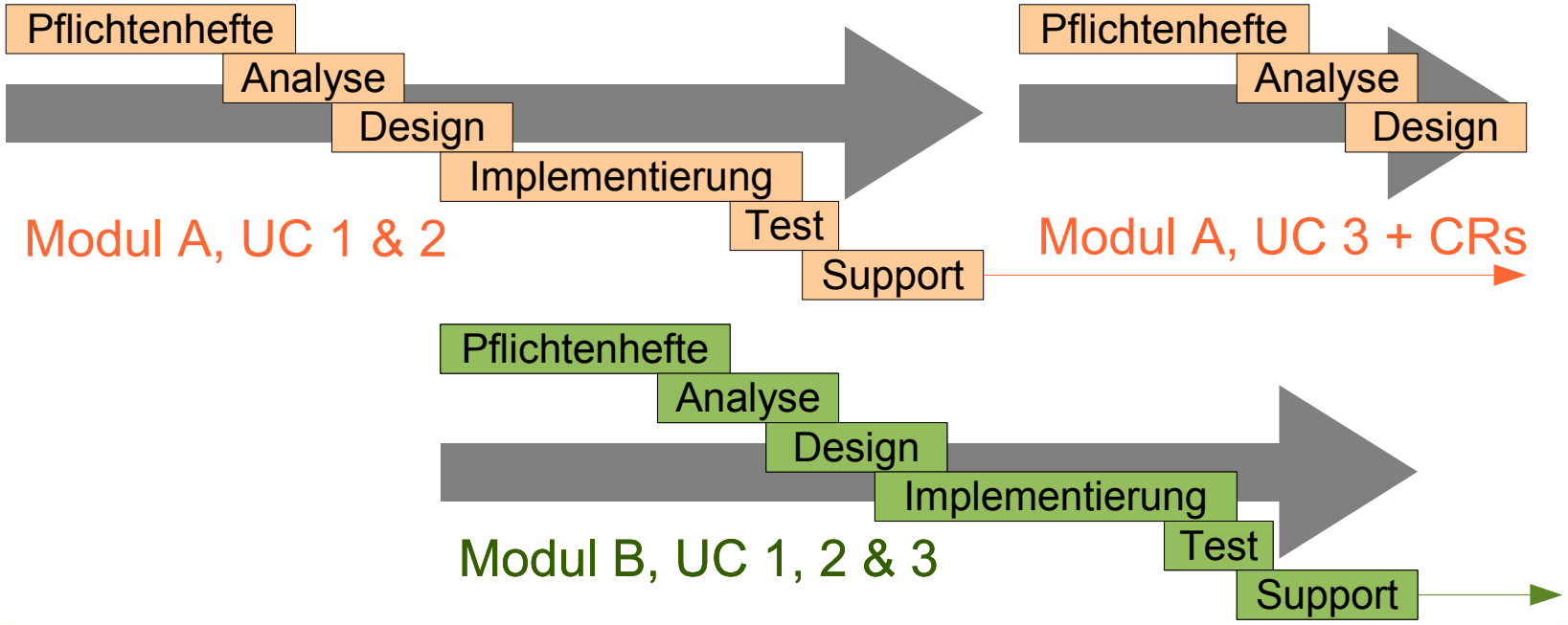
OBJECT VISION
SOFTWARE

Das Projekt

- Großer IT-Dienstleister im Finanzsektor
- Hochintegrierte Anwendung (Fat Client)
- Hausintern entwickeltes Java-Framework
- Teilprojekt Wertpapierhandel
- Hauptsächlich GUI-Abläufe und -Logik sowie Backend-Integration
- Ca. 43 MB Code in 5400 Klassen
- Ca. 35 Mitarbeiter, davon 12 Entwickler

Der Prozess

- Hausinternes Vorgehensmodell
- Paralleler Betrieb, Support und Weiterentwicklung mehrerer Versionen



Deutliche Wasserfall-Spuren...

...aber auch inkrementell

Probleme

- Überlastung des Testteams mit manuellen Tests (bes. durch verschiedene Versionen)
- Bei Zeitmangel kann nicht genug getestet werden – es muss an der Qualität gespart werden
- Immer wieder unbrauchbare Releases durch Nebenwirkungen von Änderungen – besonders in kritischen Phasen

Lösungsansatz

- Automatisierung mit Capture/Replay-Tools zweimal evaluiert und verworfen:
  - mangelnde Stabilität,
  - Keine Kontrolle über Daten im Backend
- Alternativansatz: automatisierte Entwicklertests
- Allgemeine Qualitätsverbesserung
- Vermeidung von Nebenwirkungen bei Änderungen in kritischen Phasen
- Ablauf als Teil der Entwicklungsarbeit (z.B. vor Releasebau)

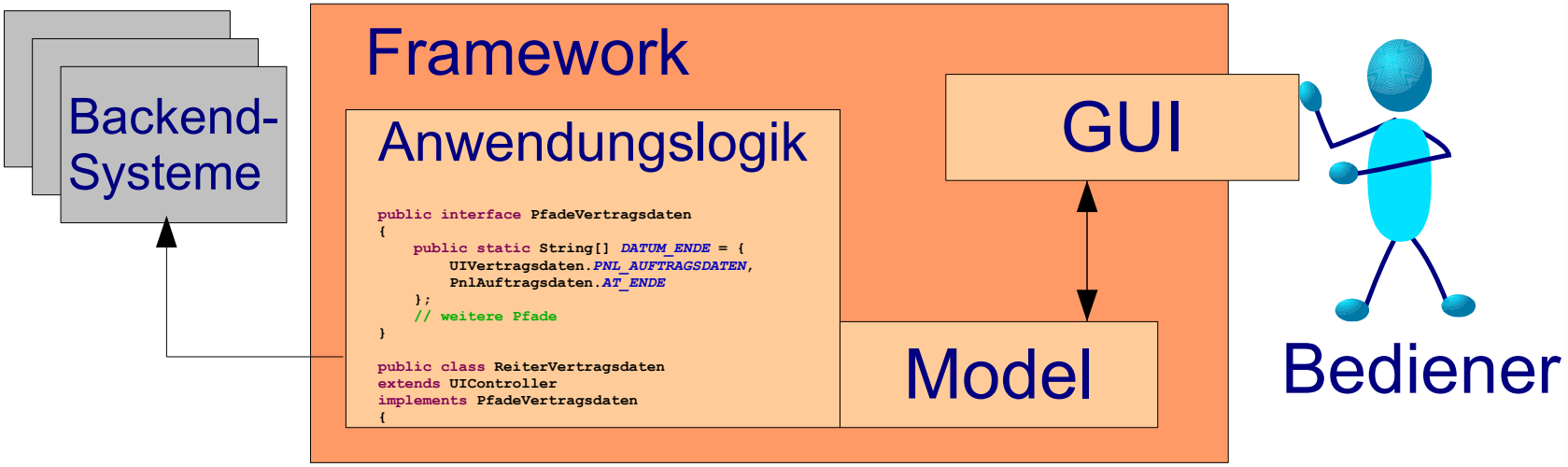
OBJECT VISION
SOFTWARE

Ablauf

1. Voraussetzungen / Umfeld
2. Umsetzung / Hürden
3. Ergebnisse / Lessons Learned

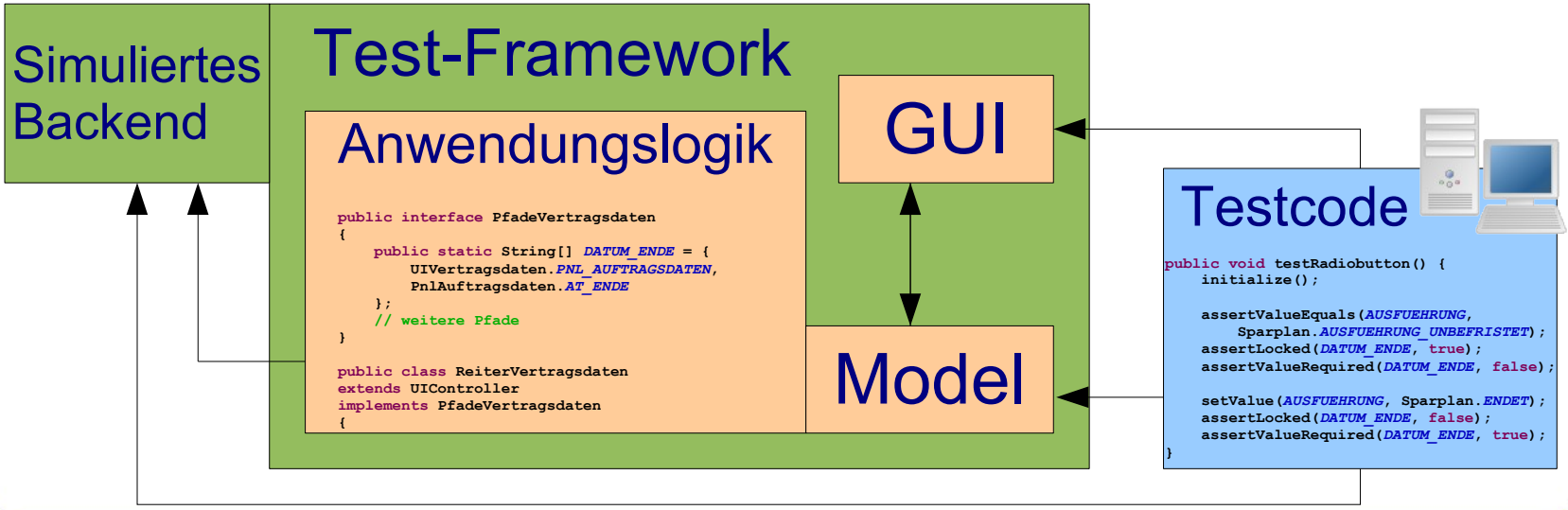
Technische Hürden

- Problem: Framework nicht auf isolierte Testbarkeit ausgelegt, fehlende Eingriffsmöglichkeiten
- Originalumgebung sehr umfangreich und dadurch langsam



Lösung

- Entwicklung eines eigenen Testframeworks
- Tests als Programmcode auf JUnit-Basis, GUI-zentriert (Manipulation & Prüfung)
- Einfache und bequeme Erstellung durch Entwickler

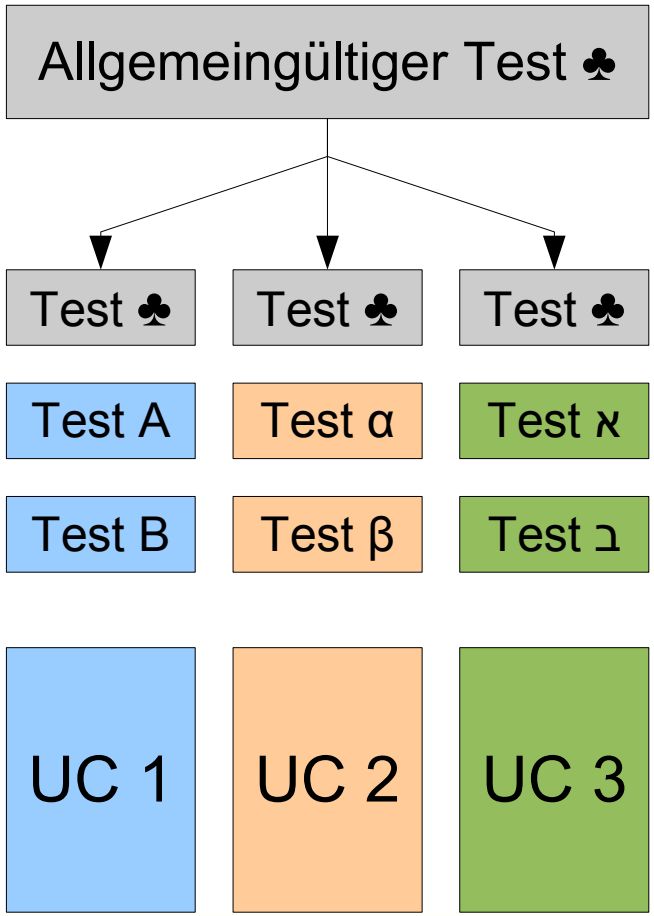


Natur der Lösung

- Näher an Integrationstests als an klassischen Unit-Tests
- Angepasst an Gegebenheiten und Bedürfnisse des Projekts
- Hohe Geschwindigkeit (1000 Testmethoden in 2 Minuten), dadurch kurzer Feedbackzyklus
- Somit gut als Teil der Entwicklungsarbeit nutzbar

Allgemeingültige Tests

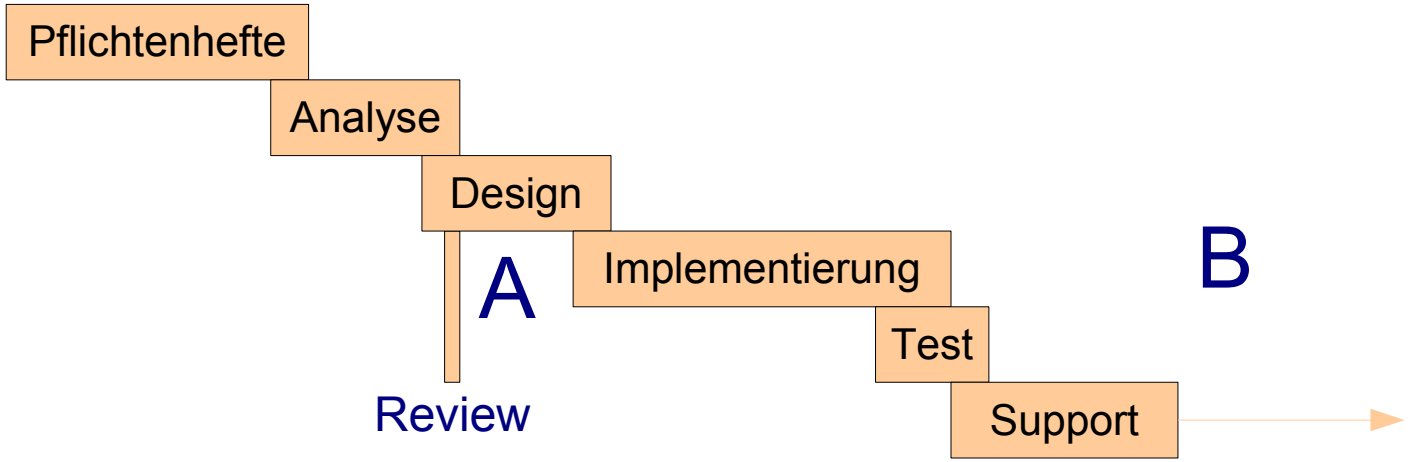
- Zusätzliche neue Möglichkeit: Überwachung von Konventionen und häufig gemachten Fehlern
- z.B. Einheitlichkeit von Tastaturshortcuts
- Allgemeingültige Tests - nur einmal zu erstellen, kein Aufwand pro Usecase



Organisatorische Hürden

- Kein etablierter Schritt im Prozess
- Keine formelle Anpassung des Prozesses auf Projektebene vorgesehen
- Managementunterstützung nur als Experiment
- Gefahr:
  - Schreiben der Tests gilt als lästig
  - wird unter Zeitdruck vernachlässigt
  - Tests werden nicht regelmäßig gestartet
  - scheiternde Tests werden ignoriert
- Am Ende ist wieder alles beim Alten

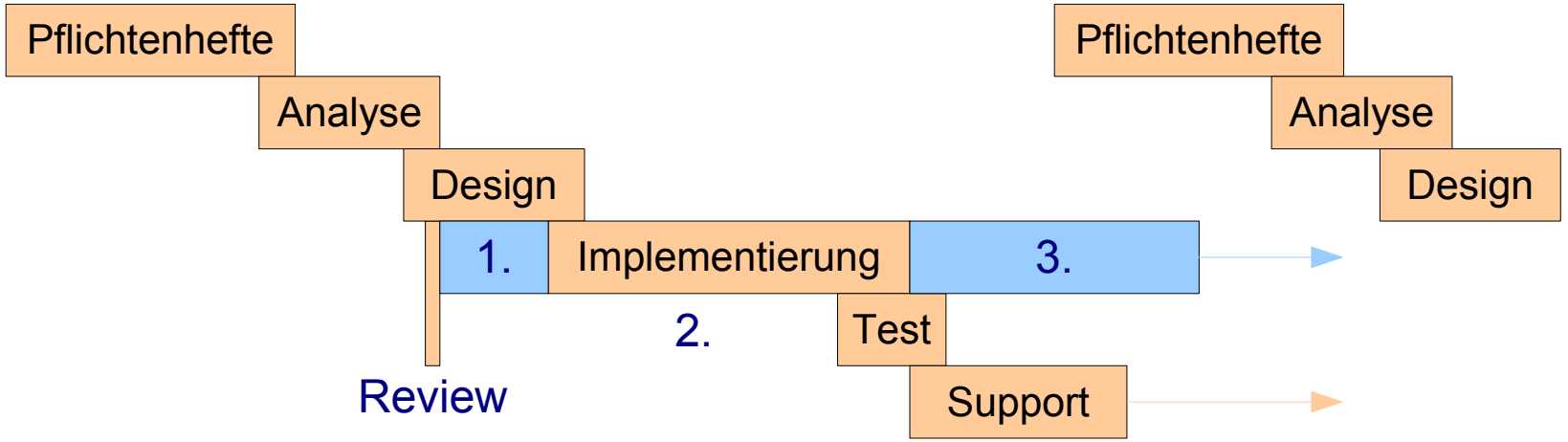
Ansatzpunkte im Prozess



A) Nach Analyseende Review durch Tester und Implementierer, noch Zeit bis Implementierungsbeginn

B) Vor Implementierungsbeginn des nächsten Releases oft wenig Arbeit für Implementierer

Einfügen in den Prozess



1. Vorläufige Testerstellung (nur GUI-Manipulation) nach Analyseende, vor Implementierungsbeginn
2. Ausimplementierung der Tests (mit Daten und Backend-Aufrufen) während der Implementierung
3. Verbesserung der Codeabdeckung in Ruhephasen

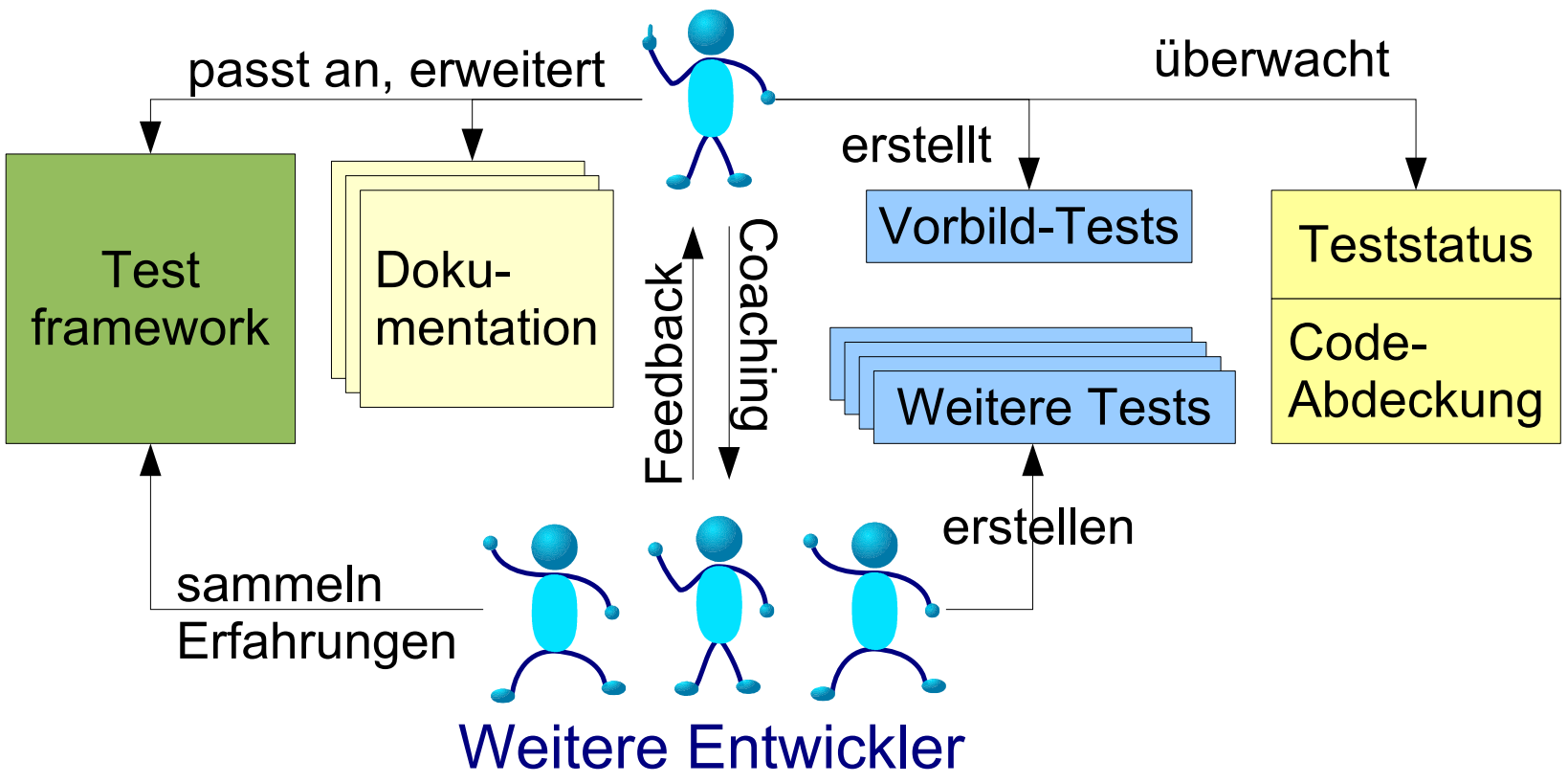
Vorteile

- Vorläufige Testerstellung auf Basis der Analysedokumente wirkt als detailliertes Review – findet mehr Unstimmigkeiten
- Tests wirken während der Implementierung als Gerüst und Erinnerung
- Ruhephasen können zur Verbesserung der Testabdeckung (d.h. der Qualität zukünftiger Releases) produktiv genutzt werden

Testfallspezifikation

- Vorgaben für Umfang der Tests:
  - Mindestens ein Testfall pro Maske
  - Abdeckung aller Pfade der Aktivitätsdiagramme (wichtigste Analysedokumente)
  - In Phase 3 dann Verbesserung auf Basis der Code-Abdeckung

Aufgabenverteilung

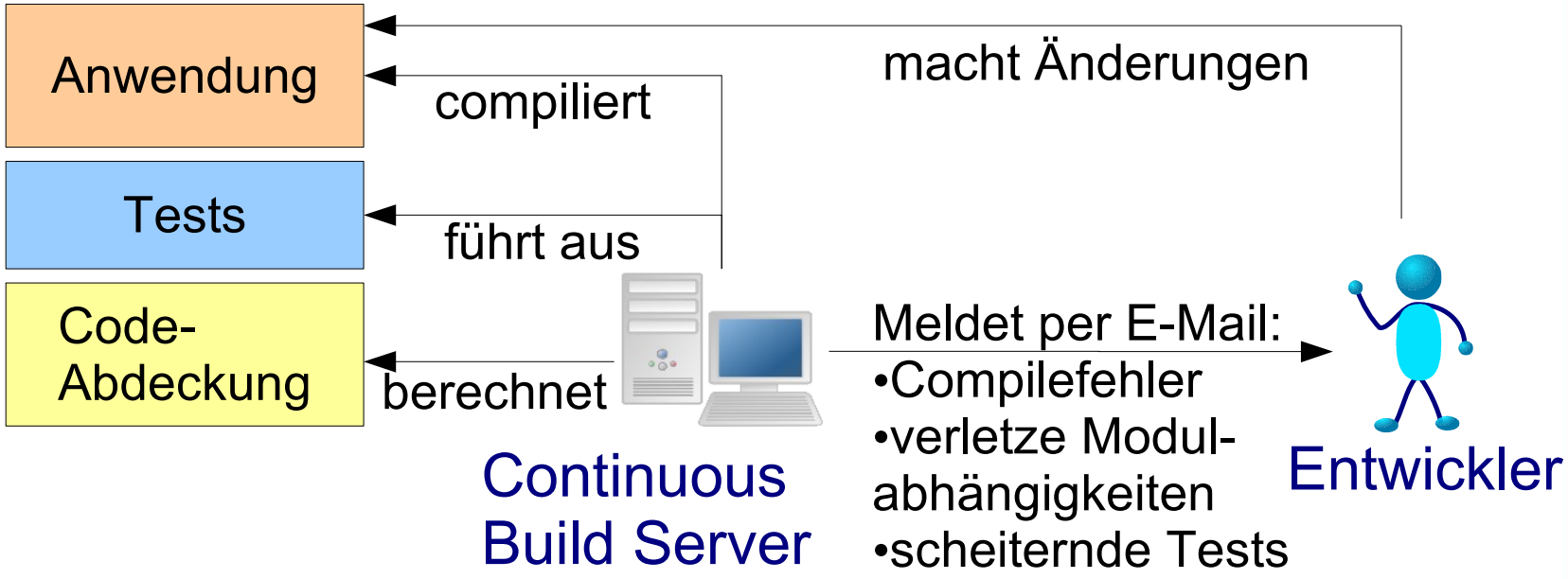


OBJECT VISION
SOFTWARE

Schleppende Adaption

- Akzeptanz durch Entwickler zunächst begrenzt, befürchtete Konsequenzen traten ein
- Besserung durch stetige Überzeugungsarbeit und Erfahrungen
- Sehr positive Reaktionen auf allgemeingültige Tests
- Schlüsselerlebnis: 2 fehlerhafte Releases an einem Tag, wäre jeweils durch vorhandene Tests abgefangen worden, die jedoch nicht gestartet wurden

Ergänzung: Continuous Build



- Alle 4 Stunden oder auf Knopfdruck
- Projektintern, unabhängig von offiziellem Release
- Verwendetes Produkt: Cruise Control

Abstimmung mit manuellen Tests

- Arbeitsteilung mit manuellen Tests verworfen, da:
  - Testframework nicht identisch zu Produktions-umgebung: keine absoluten Aussagen zur Fehlerfreiheit möglich
  - Durch automatische Tests voll abdeckbares Verhalten oft nicht deutlich von nur manuell testbarem Verhalten getrennt
  - Änderungen hier während Entwicklung möglich
  - Frühzeitige Aufteilung daher schwierig
- Hätte wesentlich weitreichendere Prozessanpassung erfordert

Ablauf

1. Voraussetzungen / Umfeld
2. Umsetzung / Hürden
3. Ergebnisse / Lessons Learned

Ergebnisse allgemein

- Generell positives Gefühl bei den Entwicklern, größere Sicherheit
- Auch bei Testern eine gefühlte Abnahme der Bugs, besonders Nebenwirkungen bei Fehlerbehebungen
- Nach positiven Erfahrungen vom Manamgement als fester Teil des Entwicklungsprozesses etabliert

Ergebnisse: Qualität

- Analyse der manuell gefundenen Bugs (je Usecase und Release) in Relation zum Umfang der automatisierten Tests
- Zahlen variieren stark, keine deutliche Korrelation
  - Entwicklerkompetenz
  - Unterschiedliche Rahmenbedingungen
  - keine Trennung zw. Frontend und Backend
  - Fehlerhafte Erfassung/Aufbereitung der Daten

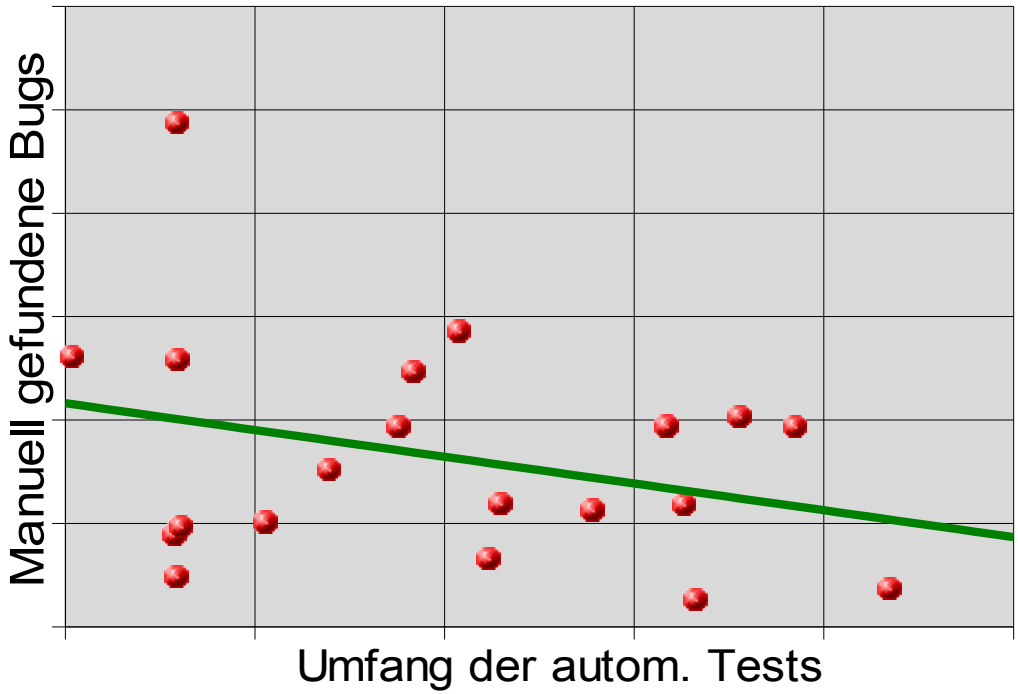
Ergebnisse: Grafik

Punkte:
- je 1 Usecase in einem best. Release

Linie:
- lineare Interpolation der Datenpunkte

X-Achse:
- autom. Tests (kB) / Code (kB)

Y-Achse:
- # Bugs (gew.) / Δ Code (kB) / # man. Tests



➔ Ucs mit bester Testabdeckung: durchschnittlich 50% weniger manuell gefundene Bugs!

OBJECT VISION
SOFTWARE

Ergebnisse: Umfrage

- Umfrage unter den Entwicklern
- Ergebnisse: Subjektive Einschätzungen
- 7 Teilnehmer

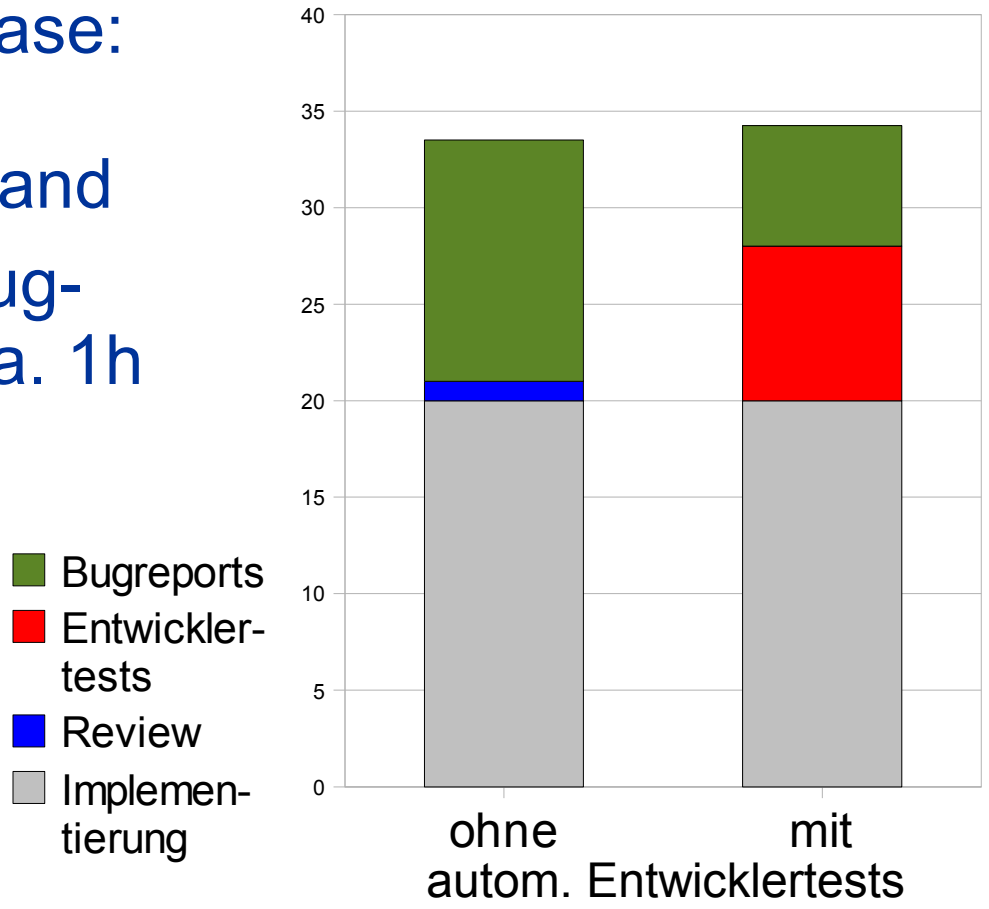
Ergebnisse: Umfrage

- Aufwand für Testerstellung relativ zur eigentlichen Entwicklung: ca. 40%
- Dafür aber Einsparungen durch besseres Review und weniger Fehler

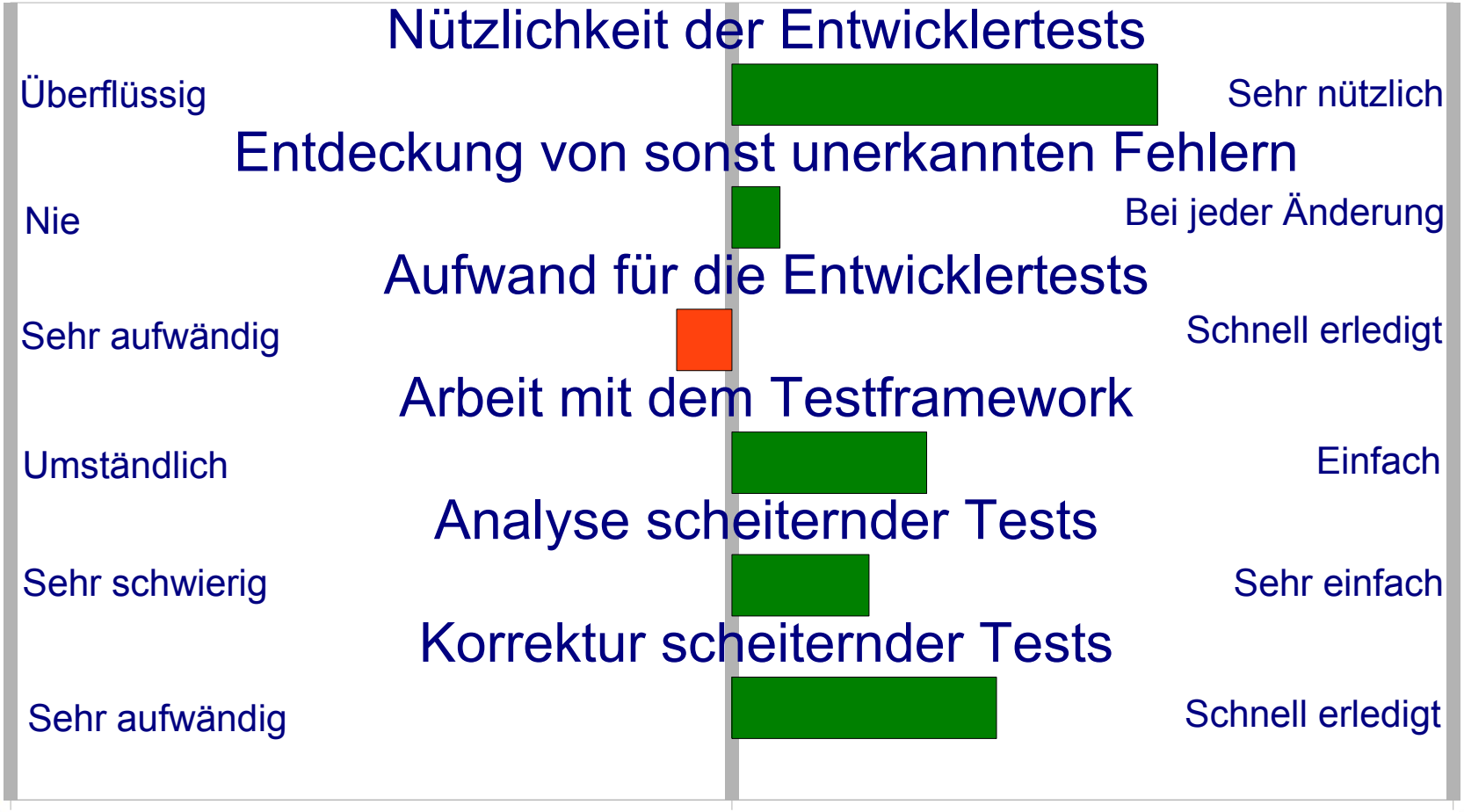
Aufwand vs. Einsparungen

- Mittelgroßer Usecase: ca. 20 PT Entwicklungsaufwand
- Normal ca. 100 Bug-Reports, jeweils ca. 1h Aufwand.

➔ Insgesamt kaum Mehraufwand!

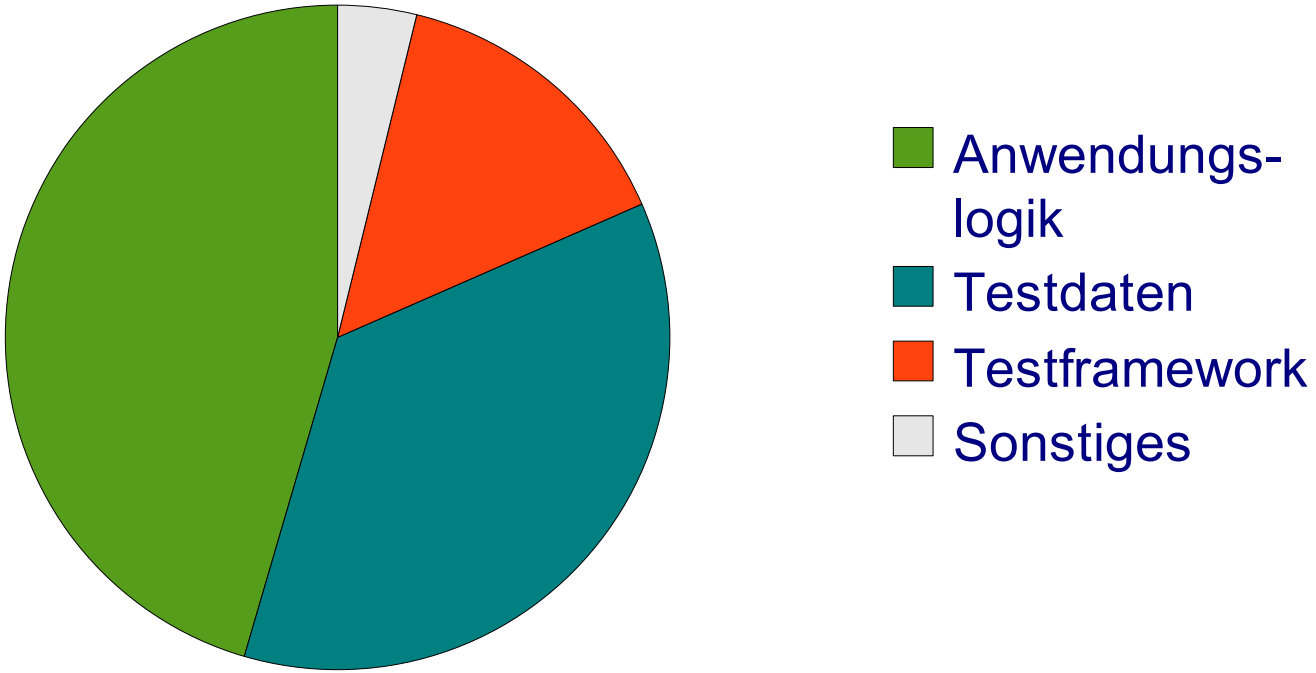


Ergebnisse: Umfrage



Ergebnisse: Umfrage

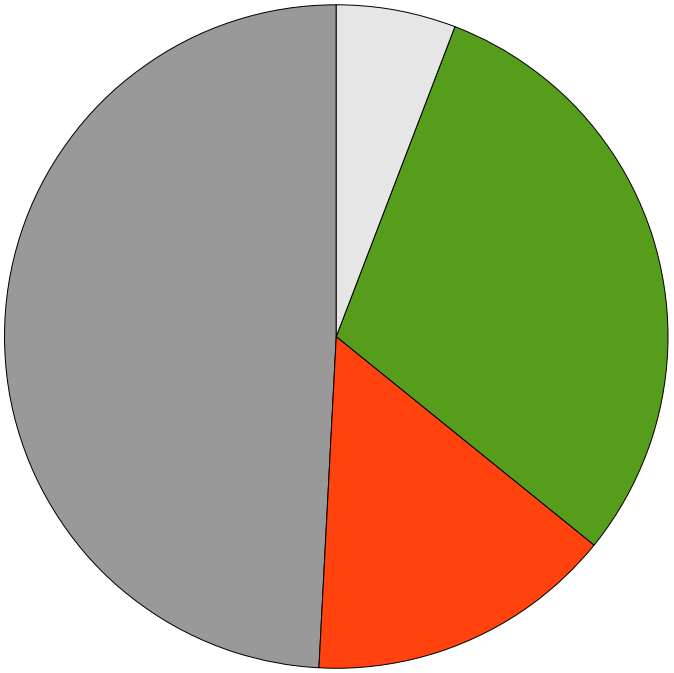
- Aufwände bei der Testerstellung



→ Hoher Aufwand für Testdaten

Ergebnisse: Umfrage

- Gründe für scheiternde Tests



→ Gute Stabilität der Tests

Was ist noch zu verbessern?

- Organisationsweite Verwendung anstreben (in Arbeit)
- Integration mit Originalframework (in Arbeit)
- Abstimmung mit Testabteilung erarbeiten
- Bessere Dokumentation und Organisation der Entwicklertests anstreben

OBJECT VISION
SOFTWARE

Lessons learned

1. Automatisierte Entwicklertests steigern Qualität deutlich und meßbar, Zusatzaufwand wird durch Einsparungen ausgeglichen
2. Sind bei Anpassung an Projekt überall möglich
3. Es muß einen motivierten Verantwortlichen geben, der die Umsetzung vorantreibt und überwacht
4. Continuous Build ist eine hochprofitable Ergänzung, wenn nicht sogar Voraussetzung für sinnvolle automatische Tests

Lessons learned

5. Schon mit kleinem Aufwand lassen sich die schwersten und häufigsten Fehler eliminieren - Möglicher Ansatz: zur Einführung nur minimale Tests (Aufruf der Funktion und allgemeine Prüfungen) – wenig Arbeit pro Usecase
6. Testerstellung läßt sich als detailliertes Review nutzen: Erzwingt Nachvollziehen statt Überfliegen der Analysedokumente. Dabei fallen wesentlich mehr Ungereimtheiten auf

OBJECT VISION
SOFTWARE

Noch Fragen?