Bremen

CG
VR

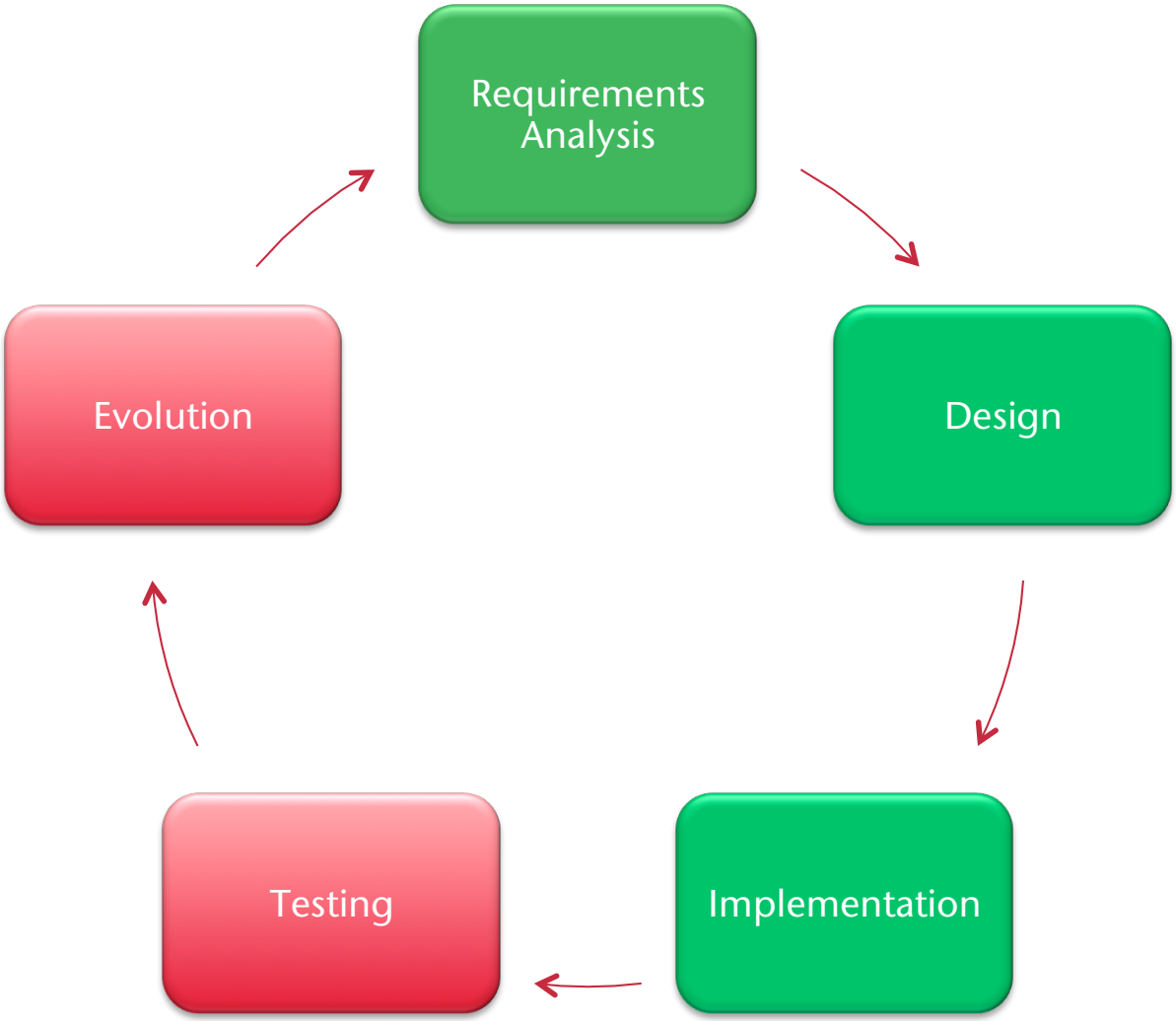
Media Engineering
Testing

R. Weller

University of Bremen, Germany

cgvr.cs.uni-bremen.de

Der Software Development-Lifecycle



Zur Erinnerung: Software-Projekt-Desaster

- Beispiel: Stadionüberwachung KSC
  - Tests wurden nach Fanprotesten abgesagt

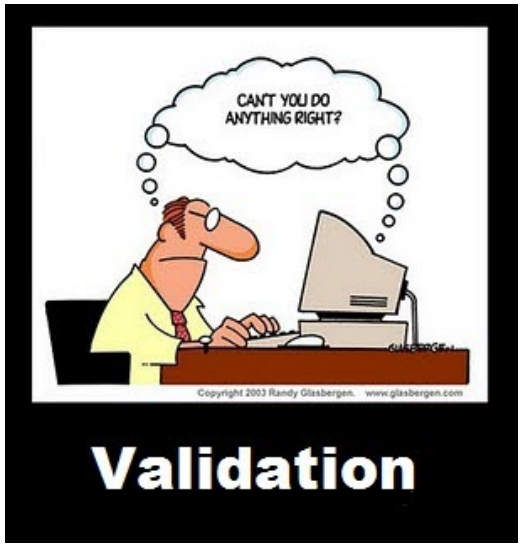
- Beispiel Ariane 5
  - Stürtzte wegen Softwarefehler ab

Der verursachende Code (Ada) ...

```
...
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
  ...
begin
  declare
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get( vertical_veloc_sensor );
    sensor_get( horizontal_veloc_sensor );
    vertical_veloc_bias := integer( vertical_veloc_sensor );
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception
    when numeric_error => calculate_vertical_veloc();
    when others => use_irs1();
  end;
end irs2;
```

Was wollen wir testen?

- Validierung: Are we doing the right thing?
  - Ist das, was wir machen, auch das, was gewünscht wird?
    - Vom Auftraggeber (Pflichtenheft)
    - Vom Kunden/Markt
- Verifikation: Are we doing things right?
  - Haben wir einen bestimmten Entwicklungsschritt richtig durchgeführt?
    - Z.B. Stürzt die Software ständig ab?
    - Ist die GUI bedienbar?



Warum testen wir?

- Validierungstests
  - Zeigen dem Kunden (aber auch dem Entwickler), dass die Anforderungen erfüllt wurden
    - Für jede Anforderung im Pflichtenheft sollte es einen Testfall geben
  - Bei generischen Produkten (Server, Datenbanken, Libraries): Tests aller Systemfunktionen sowie deren Kombination
- Fehlertests
  - Spüren Situationen auf, in denen sich die Software falsch verhält
  - Können obskur gehalten sein und müssen nichts mit der normalen Verwendung des Systems zu tun haben
- Grenzen oft fließend
  - Validierungstests zeigen oft Fehler im System an
  - Einige Fehlertests können erfüllte Anforderungen anzeigen

Wer testet?

1. Entwicklertests
   - Finden von Programmierfehlern während der Entwicklung
   - Durchgeführt von den Entwicklern selbst
2. Freigabetests
   - Optimalerweise separates Testteam
   - Test einer vollständigen Version des Systems ehe es freigegeben wird
3. Benutzertests
   - Mögliche Benutzer testen das System in eigener Umgebung
     - Benutzer können z.B. (interne) Marketinggruppen sein
     - Oder auch Experten die befragt werden
   - Abnahmetests durch den Kunden als spezieller Benutzertest

Soziologie des Testens

- Testen wird häufig (aber zu unrecht!) als niedere Arbeit angesehen
  - => Neulinge werden zum Testen abkommandiert
  - Problem:
    - Tester brauchen umfassendes Systemverständnis (Anforderung, Entwurf, Implementierung)
    - Tester brauchen darüber hinaus Wissen über Prüftechniken
- Entwickler führen selbst Abnahmetests durch
  - Problem:
    - Entwickler haben eine Lesart (von möglicherweise mehreren) der Spezifikation verinnerlicht
    - Denkfehler in der Implementierung wiederholen sich bei Erstellung von Testfällen
- Kritik von Testern am Produkt wird als Kritik am Entwickler aufgefasst

Dokumentation von Tests

- Testplan
  - Projektplan fürs Testen
  - Inklusive Spezifikation von Testfällen
- Testprotokoll (auch Testvorfallbericht)
  - Ergebnisse des Tests und Unterschiede zu erwarteten Ergebnissen
- Testübersicht
  - Auflistung aller Fehler (entdeckte und noch zu untersuchende)
  - Erlaubt Analyse und Priorisierung aller Fehler und deren Korrekturen

Dokumentation von Tests

- Testplan (nach IEEE Std 829-1998)

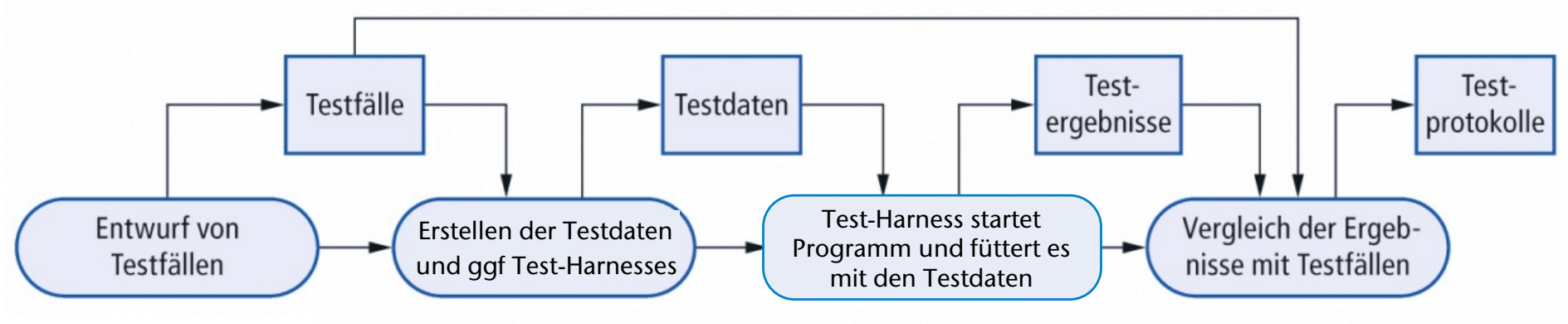
1. Einführung
2. Systemüberblick
3. Merkmale die (nicht) getestet werden müssen
4. Abnahmekriterien
5. Vorgehensweise
6. Aufhebung und Wiederaufnahme
7. Zu prüfendes Material (Hardware-/Softwareanforderungen)
8. Testfälle
9. Testzeitplan: Verantwortlichkeiten, Personalausstattung und Weiterbildungsbelange, Risiken und Schadensmöglichkeiten, Zeitplan

Testfallspezifikation

- Testfallbezeichner
  - eindeutiger Name des Testfalls; am Besten Namenskonventionen benutzen
- Testobjekte
  - Komponenten (und deren Merkmale), die getestet werden sollen
- Eingabespezifikationen
  - Konkrete Eingabedaten
- Ausgabespezifikationen
  - erwartete Ausgaben
- Umgebungserfordernisse
  - notwendige Software- und Hardware-Plattform
- Besondere prozedurale Anforderungen
  - Einschränkungen wie Zeitvorgaben, Belastung oder Eingreifen durch den Benutzer
- Abhängigkeiten zwischen Testfällen

Testprotokoll

- Dokumentiert:
  - Welche Merkmale wurden getestet
  - Wurden sie erfüllt?
  - Bei Störfall: Wie kann er reproduziert werden
- Wichtig: Test bedeutet nicht Fehlersuche bzw –korrektur!



Testen und dann läuft's schon?

- Wichtig: Tests können nur die Anwesenheit von Fehlern aufzeigen, aber nicht ihre Abwesenheit!
  - Testen ist wichtig, garantiert aber kein fehlerfreies Produkt
- Weitere Fehlervermeidungsstrategien sind ebenso wichtig
  - Fehlervermeidung
    - Z.B. Entwicklungsmethoden, statische Analysen,...
  - Fehlertoleranzen
    - Behandlung von Fehlern zur Laufzeit

Entwicklertests

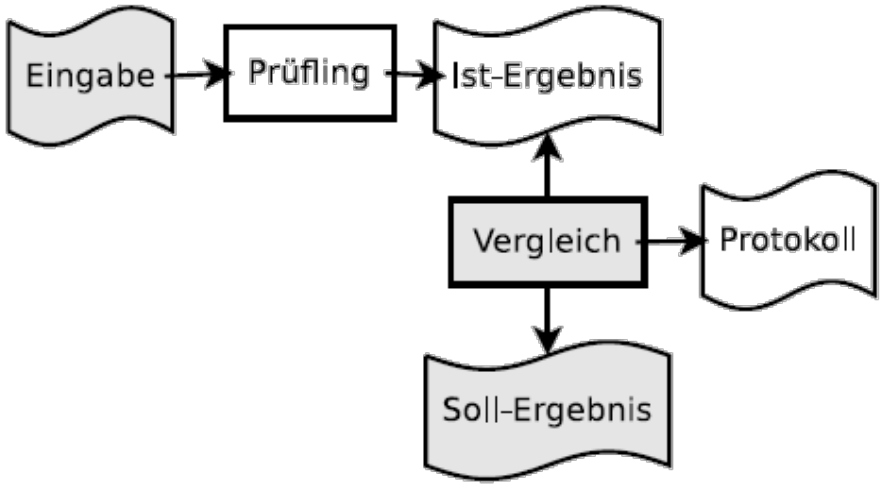
- Testaktivitäten die vom selben Team durchgeführt werden, das auch das System entwickelt
- Arten von Entwicklertests
  - Unit Tests (auch Modultests)
    - Test der Funktionalität einzelner Programmeinheiten (Methoden, Klassen,...)
  - Integrationstests
    - Module werden zu größeren Komponenten zusammengesetzt
    - Testen der Komponentenschnittstellen
  - Systemtests
    - Testen des Systems als Ganzes (wenn alle oder einige Komponenten integriert wurden)
    - Testen der Interaktionen zwischen den Komponenten

Unit Tests

- Testen von kleinsten funktionalen Einheiten: Klassen, Methoden
- Unit-Tests sollen zweierlei leisten:
  - Wenn die Units korrekt verwendet werden, sollen sie das gewünschte Ergebnis liefern
  - Falls die Unit fehlerhaften Input enthält, soll abgefangen werden (auch bei nicht korrekter Verwendung sollen keine Fehler auftreten)
- Prinzipiell:
  - Für Methoden:
    - Füttere Methode mit (möglichst vielen) Eingaben und schaue ob das gewünschte Ergebnis rauskommt
  - Für Klassen:
    - Teste alle Methoden der Klassen
    - Versetze Objekt in (möglichste alle) Zustände und schaue, ob es sich wie gewünscht verhält (also: simuliere alle Zustandsänderungen)
    - Setze alle Attributwerte und frage sie ab

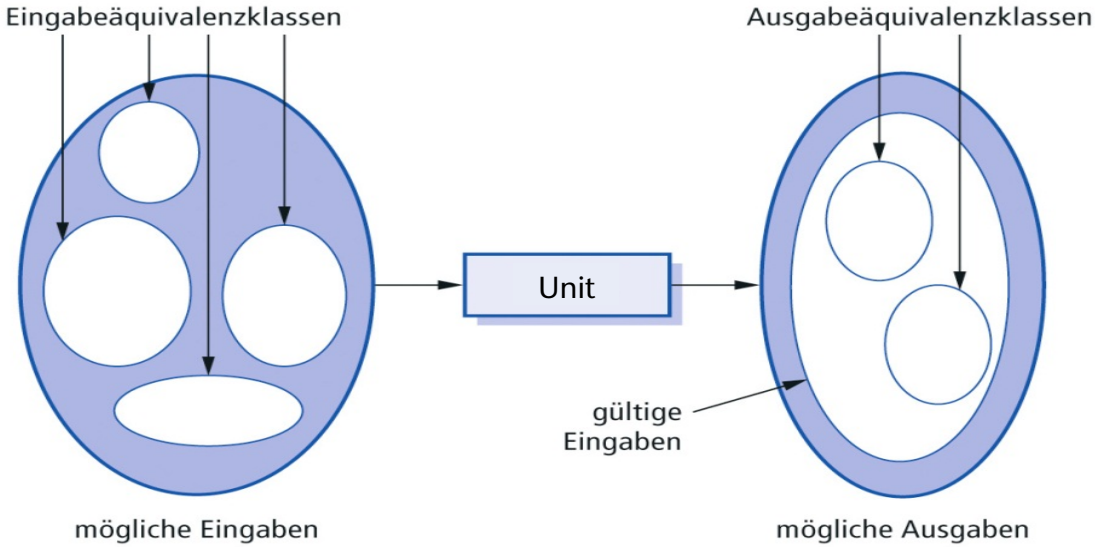
Typen von Unit Tests

1. Black-Box-Tests: Testfall wird nur mit Kenntnis der Schnittstellenbeschreibung entworfen (Implementierung unbekannt)
2. White-Box-Tests: Testfall wird mit Kenntnis der Implementierung entworfen



Black-Box-Test: Äquivalenztest

- Problem: Testen *aller* Eingabe- und Ausgabeparameter oft zu aufwendig
- Beobachtung: Eingegebene Daten und ausgegebene Resultate verteilen sich oft in verschiedene Klassen mit gemeinsamen Eigenschaften
- Idee: Identifiziere diese Klassen anhand der Spezifikation(!) und wähle jeweils einen Repräsentanten zum Testen aus



Beispiel für Äquivalenztest

```
int numberOfDays( int year, int month )
{
    int numberOfDays;
    if( year < 1 )
        throw std::invalid_argument( "InvalidYear" );
    if( month < 1 || month > 12 )
        throw std::invalid_argument( "InvalidMonth" );
    if (month == 4 || month == 6 || month == 9 || month == 11)
      numberOfDays = 30;
    else if (month == 2)
    {
      bool isLeapYear = (year % 4 == 0 && year % 100 != 0);
      if (isLeapYear)
        numberOfDays = 29;
      else
        numberOfDays = 28;
    }
    else
      numberOfDays = 31;
  return numberOfDays;
}
```

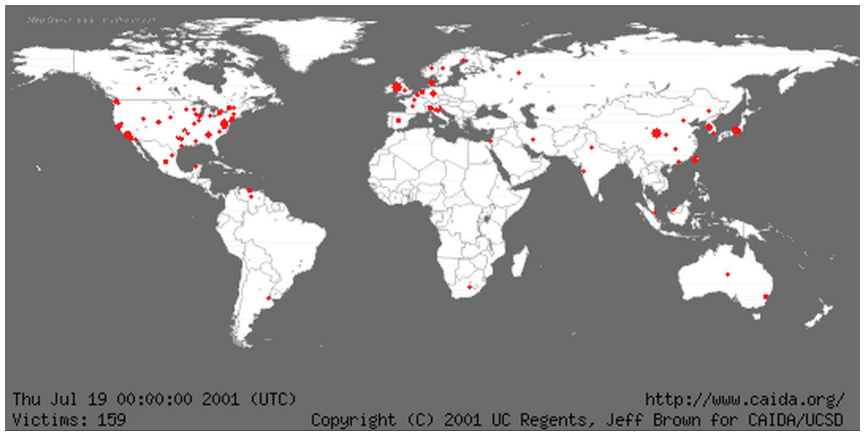
Beispielspezifikation

- `numberOfDays(year, month)` liefert die Tage eines Monats wie folgt:
  - month$\in\{1,3,5,7,8,10,12\}\Rightarrow 31$
  - month$\in\{4,6,9,11\}\Rightarrow 30$
  - `year` ist Schaltjahr und month$==2\Rightarrow 29$
  - `year` *ist kein Schaltjahr und month*$==2\Rightarrow 28$
- Fehlerfall
  - month < 1 oder month >12 ⇒ `throw InvalidMonth`
  - `year`< 0 ⇒*throw InvalidYear*

Beispieltestfälle für Äquivalenztest

Äquivalenzklasse	Monat	Jahr	Soll
31-Tage-Monat, Nicht- /Schaltjahr	7	1904	31
30-Tage-Monat, Nicht- /Schaltjahr	6	1904	30
Februar, Nicht-Schaltjahr	2	1901	28
Februar, Schaltjahr	2	1904	29
Monat inkorrekt, Jahr korrekt	17	1901	`InvalidMonth`
Monat korrekt, Jahr inkorrekt	7	-1904	`InvalidYear`

- Wichtig: teste Fehlerfälle und "bogus input"-Fälle !
- Beispiel: Buffer Overflow
  - Extrem häufiger und gefährlicher Bug, der zu Security Leaks führt
  - Verursacht 60% der CERT Advisories (= Security-Warnungen)
- Beispiel "Code Red Worm", 19. Juli 2001:
  - Nutzte Leak in Microsoft's IIS web server
    - Virus schickte spezielle, sehr lange URL (100,000 Zeichen) an Web-Server ⟶ buffer overflow ⟶ web server führte Code aus der URL mit Root-Rechten aus
  - 2.5 Milliarden US$ Schaden (Produktionsverlust, Arbeitszeit für Clean-Up)



Ausbreitung von 19. Juli bis 20. Juli, 2001

Wichtiger Spezialfall: Grenztest

- Wieviele Pfeiler hat das Brandenburger Tor?

- Beobachtung: „Off-by-One“-Fehler kommen häufig vor
- Idee: teste Grenzen der Äquivalenzklassen (Randbereiche, Sonderfälle)

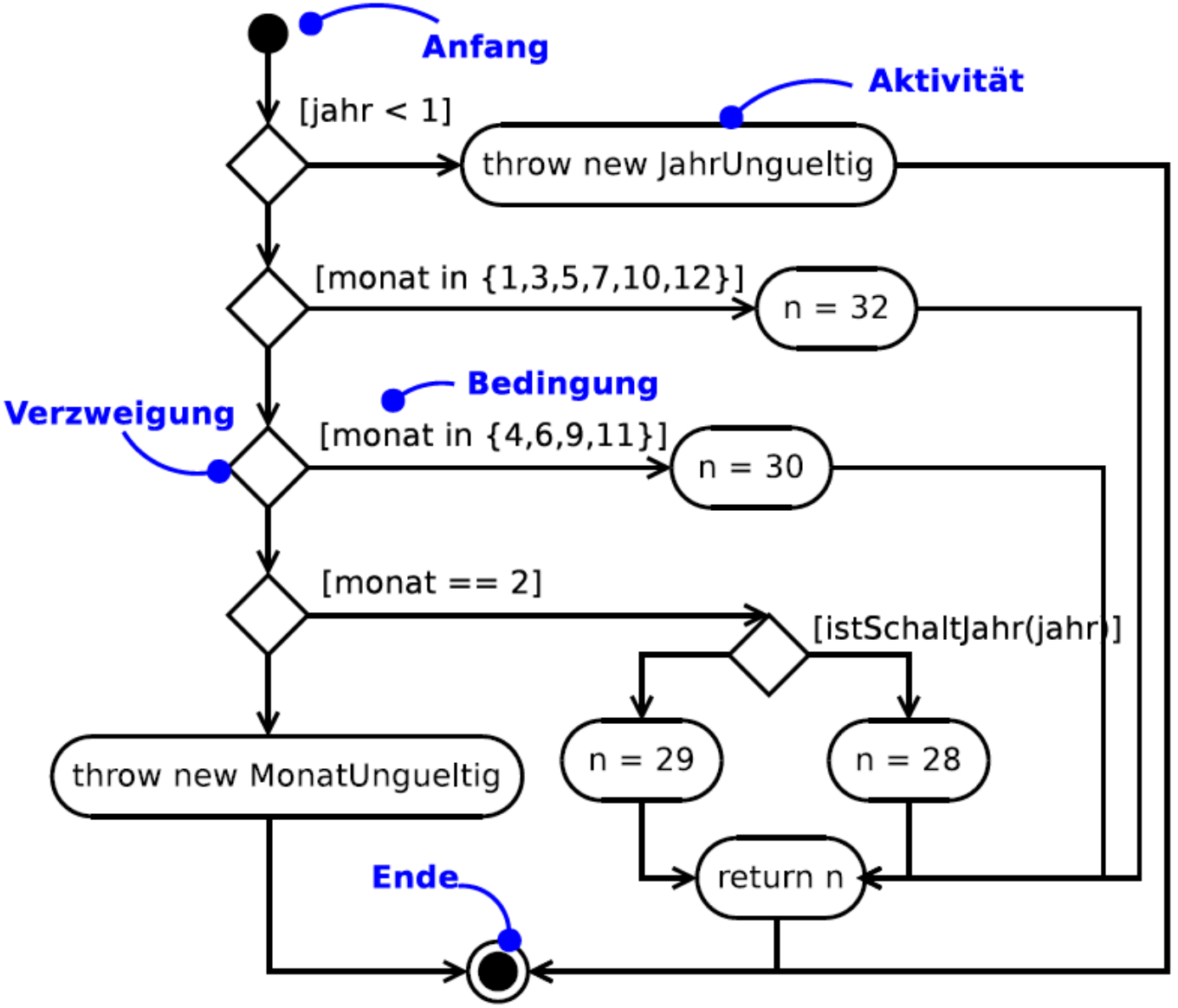
Beispieltestfälle für Grenztest

Äquivalenzklasse	Monat	Jahr	Soll
Erster gültiger Monat	1	1234	31
Letzter gültiger Monat	12	1234	31
Erster ungültiger Monat	0	1234	`InvalidMonth`
Nächster gültiger Monat	13	1234	`InvalidMonth`
Erstes gültiges Jahr	1	0	31
Letztes gültiges Jahr	1	Max Int	31
Erstes negatives Jahr	1	-1	`InvalidYear`
Erstes Schaltjahr	2	4	29
...	...	...	...

White-Box-Test: Pfadtest

- Ziel: Testfälle sollen alle Code-Teile testen (code coverage)
- Idee: Konstruiere Testfälle, die jeden möglichen Pfad mindestens einmal ausführen

Beispiel Pfadtest



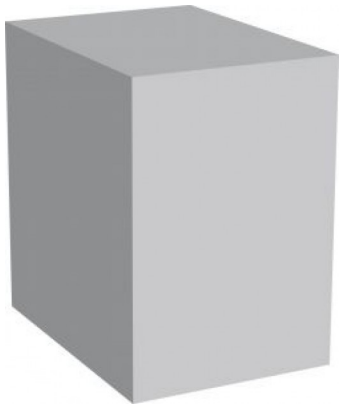
White-Box vs Black-Box

Eigenschaft	Black-Box-Test	White-Box-Test
Test auf Basis von	Schnittstellenspezifikation	Source-Code
Wiederverwendung bei Änderungen im Code	Ja	Eingeschränkt
Geeignet für Testart	Unit Test, Integrationstest, Systemtest	Unit Test
Finden der Fehler aufgrund von	Abweichungen zur Spezifikation	Eher Coding-Fehler

- Nicht entweder-oder, sondern sowohl-als-auch!

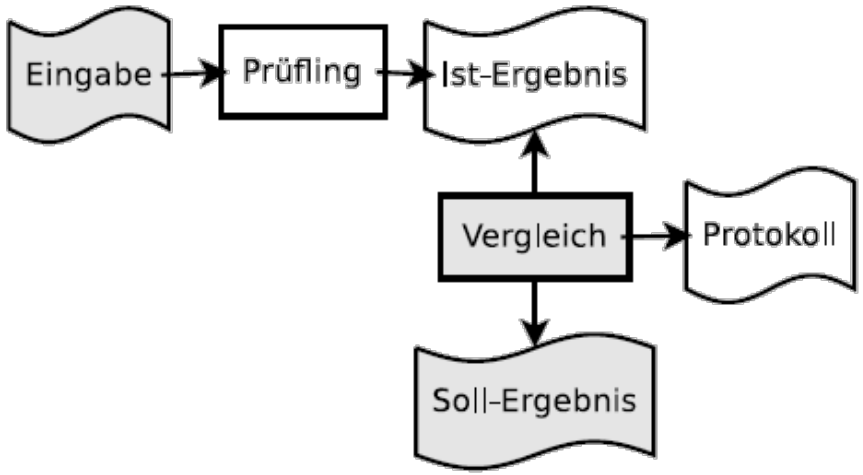
White + Black = Grey-Box-Tests?

- Idee: Schreibe Test vor der eigentlichen Implementation (Test-First-Programmierung bzw. *test-driven development*)
- Anschließende Implementierung, bis alle Tests bestanden sind
  - Dann noch eventuelles Refactoring und Schönschreiben
- Verhindert, dass um Fehler herum getestet wird
  - Betriebsblindheit, wenn Tester und Programmierer dieselbe Person ist
- Oft Bestandteil agiler Softwareentwicklung
  - Dazu später mehr



Automatisierung

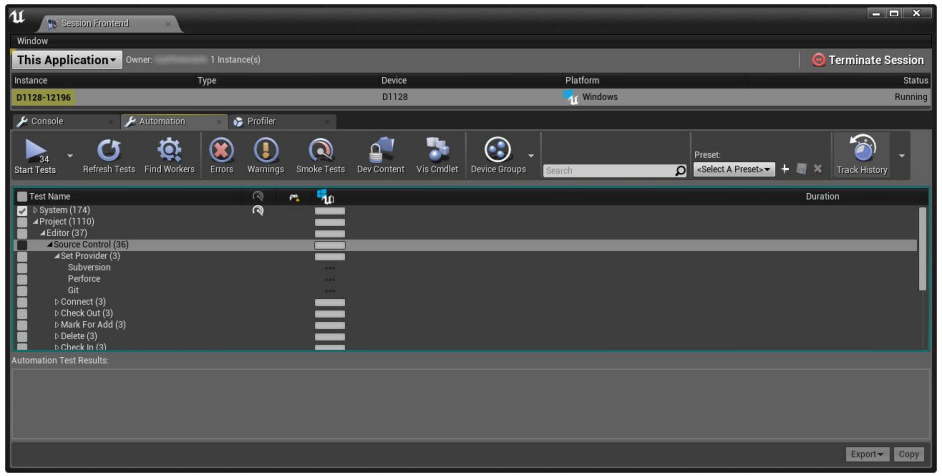
- Zur Erinnerung: Grundvorgehen bei Unit-Tests
  - Definiere Eingabe und Soll-Ausgabe
  - Teste Unit mit Eingabe
  - Vergleiche Ist-Ausgabe mit Soll-Ausgabe



- Vieles davon ist ziemlich generisch und wiederholt sich oft für jede Unit
  - Schreiben von Test-Harnesses
  - Ausführen der Tests
  - Vergleich zwischen Ist- und Soll-Werten
  - => Unit-Test-Frameworks nehme einem diese langweilige Arbeit ab

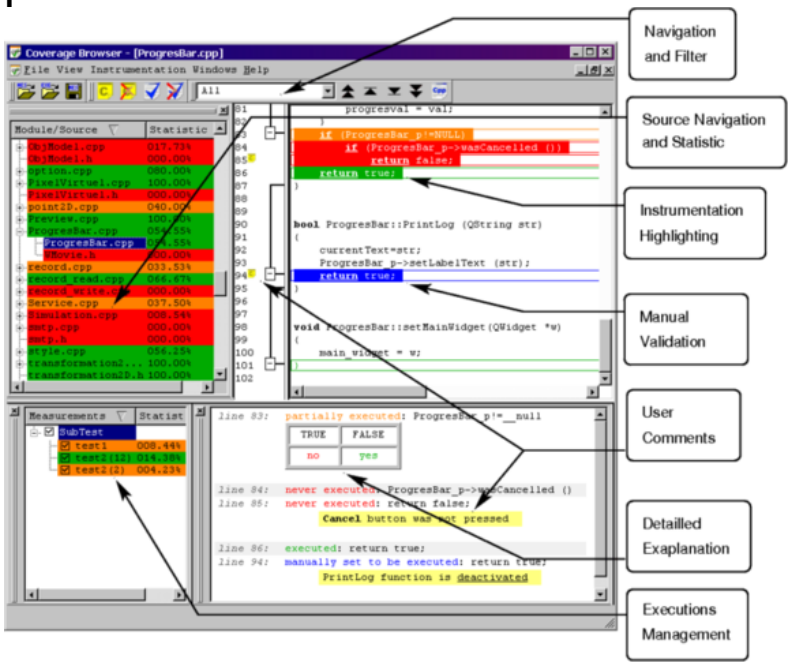
Unit-Test-Frameworks

- Funktionalität:
  - Aufbau, bei dem das System mit den definierten Testfällen initialisiert wird
  - Aufruf der zu testenden Methoden
  - Vergleich mit den vorher definierten Soll-Ergebnissen und Bewertung der Ausgabe
- Beispiele:
  - JUnit (Java)
  - CppUnit (Portierung von JUnit auf C++)
  - Populäre Libraries wie Qt und Boost bieten Unit-Test-Funktionalität
  - Die Unreal Engine unterstützt Unit-Tests (und einige weitere Testarten) mit dem "Automation System"



Weitere Tools

- Code Coverage-Tools
  - Unit-Tests sollten idealerweise 100% des Codes abdecken, mindestens aber 70-80%
  - Code-Coverage-Tools messen, wieviel Code durch Unit-Tests überprüft wird
  - Funktionsweise: Zur Compile-Zeit werden Marker mit in den Code kompiliert (code instrumentation)
  - Beispiel: CoverageMeter (C++)
- Mocking-Tools
  - Simulieren Schnittstellen, die noch nicht implementiert sind
  - Beispiel: TypeMock



Entwicklertests: Integrationstests

- Test von Zusammenfassungen von Modulen zu größeren Komponenten und deren Kommunikation untereinander
  - Auch Schnittstellentests genannt
- Beispiele für Schnittstellen zwischen Komponenten
  - Parameterschnittstellen (Übergabe von Daten)
  - Schnittstellen mit gemeinsamem Speicher (Beispiel: Eine Komponente schreibt Daten in Speicherbereich, eine andere liest sie aus)
- Mögliche Fehler bei Verwendung von Schnittstellen
  - Falsche Verwendung der Schnittstelle
    - Beispiel: Parameter vom falschen Typ, falsche Anzahl Parameter,...
  - Schnittstellenmissverständnisse
    - Beispiel: Binäre Suche auf ungeordnetem Feld
  - Zeitabstimmungsfehler
    - Beispiel: Produzent und Konsument arbeiten mit unterschiedlicher Geschwindigkeit

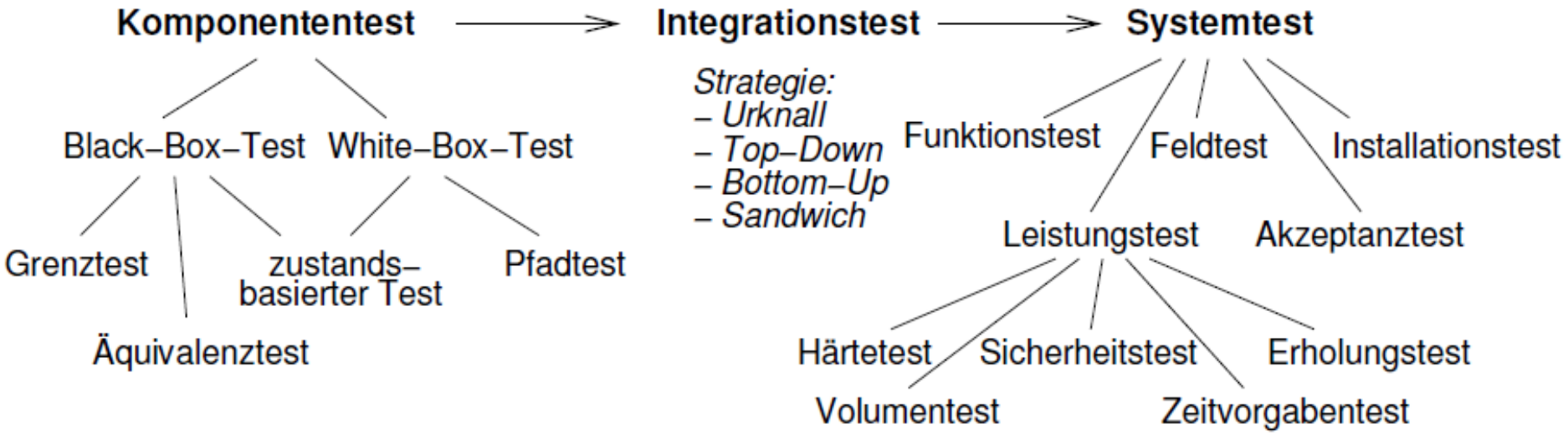
Strategien für Integrationstests

- Urknalltest: Alle Komponenten werden einzeln entwickelt und in einem Schritt integriert
  - Problem: Erschwert Fehlersuche
- Bottom-Up: Integration erfolgt inkrementell in umgekehrter Richtung zur "Benutzt"-Beziehung (Vgl UML-Diagramme)
  - Problem: Fehler in oberster Schicht werden spät entdeckt
- Top-Down: Integration erfolgt in Richtung "Benutzt"-Beziehung
  - Problem: Unit-Tests der unteren Schichten stehen noch nicht zur Verfügung, man weiß also nicht, ob diese fehlerfrei sind

Entwicklertests: Systemtests

- Interne Vorbereitung auf abschließenden Freigabetest
- Test des Gesamtsystems
  - Funktionale Tests (Sind alle Funktionen aus dem Pflichtenheft enthalten und funktionieren sie richtig?)
  - Nicht-Funktionale-Tests (z.B. Performancetest, Sicherheitstest, Erholungstest, Volumentest,...)
- Im Gegensatz zum Integrationstest (und natürlich auch Unit Test) findet der Systemtest nicht in der Entwicklungsumgebung, sondern in einer der späteren Produktivumgebung ähnlichen Testumgebung statt
  - Diese sollte die Produktivumgebung möglichst realitätsnah simulieren
- Wird, insbesondere bei größeren Projekten, von eigenem Testteam durchgeführt

Zusammenfassung der Entwicklertestarten

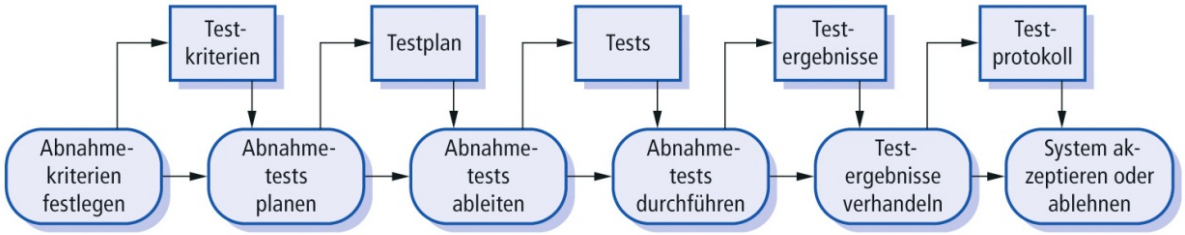


Freigabetests

- Ziel: Den Anbieter des Systems (nicht den Kunden!) von der Qualität zu überzeugen
- Sollte nicht vom Entwickler durchgeführt werden, sondern von einem separaten, nicht an der Entwicklung beteiligten Team
- Grundsätzlich zwei Vorgehensweisen:
  - Anforderungsbasiertes Testen
    - Systematisches Abhaken der (quantifizierbaren) Anforderung im Pflichtenheft
  - Szenariobasiertes Testen
    - Definition typischer Benutzerszenarios
    - Durchspielen der Szenarios in der Rolle des Benutzers

Benutzertests

- Nach Entwickler- und Freigabetests (beim Hersteller) ist das Produkt fast bereit für die Auslieferung. Vorher sollte es aber noch vom zukünftigen Kunden getestet werde, da Entwickler gerne etwas übersehen
- Prinzipiell drei unterschiedliche Typen
  - Alphatests
    - Benutzer arbeiten mit Entwicklern zusammen und testen System schon in der Entwicklungsumgebung
  - Betatests
    - Benutzer testen das System in eigener Umgebung und geben Feedback an die Entwickler
  - Abnahmetests
    - Der Kunde testet das System und entscheidet, ob er es abnehmen kann



Betatests für Spiele

- Insbesondere bei Massenprodukten wie Computerspielen werden Betatests immer populärer
- Vorgehen: Interessierte Nutzer melden sich per Internet für Betatests an
- Vorteile:
  - Für Spieler: kommen früher an ersehnte Spiele
  - Für Hersteller
    - Relativ günstig
    - Große Basis an unterschiedlichen Hardwarekonfigurationen
    - Insbesondere bei (Massively) Multiplayer-Online-Games eigentlich alternativlos
    - Neben klassischer Fehlersuche: Gleichzeitig Feedback über softere Kriterien wie Game Balancing, Spaßfaktor, Lernkurven, Usability, ...

Testen der „soften“ Faktoren

- Im Gegensatz zu z.B. Units (z.B. Funktionen, Klassen), lässt sich menschliche Interaktion mit dem Produkt (z.B. Software) kaum maschinell testen
  - => Man benötigt Menschen zum Testen
- Prinzipiell zwei Vorgehensweisen:
  - Benutzertests
  - Expertenbasierte Evaluation

Expertenbasierte Evaluation

- Heuristische Evaluation
  - Überprüfen ob Standards und Guidelines eingehalten wurden
  - Beispiele: Farbgebung bei Webseiten, magische 7, Gestaltgesetze, Fitt‘s und Hicks‘-Gesetze
    - IEEE Guidelines für Usability,
  - Man sollte immer mehrere Experten befragen
- Cognitive Walkthrough
  - Definition von Benutzerszenarien und Benutzercharakteristiken
  - Experte spielt Beispielszenarien durch und versetzt sich dabei in die Rolle des Beispielbenutzers
  - Am Ende gibt der Experte Hinweise, was noch nicht funktioniert

Definition von Szenarios

- Sollten sich an spätere „echte“ Szenarien anlehnen
- Szenarien aus der Design-Phase (oder aus der Requirements-Engineering-Phase) können verwendet werden
  - Eventuell kürzen, falls zu lang
  - Benötigen eventuell Vorwissen
- Training sollte vermieden werden (falls es nicht beim Endprodukt auch vorgesehen ist)
- Bei der Auswahl der Beispielszenarios sollte man Bias vermeiden!
- Ebenso sind zu kleinteilige Aufgaben zu vermeiden (bei denen die Aufgabenstellung schon die Lösung vorwegnimmt)

Beispielszenarien

Table 1.1. Examples of Tasks Used in Our Usability Testing

Open-ended Tasks	Directed Tasks
Find the symptoms of swine flu and what you should do to avoid getting sick.	Use yelp.com to find reviews of the San Francisco restaurant Absinthe.
Check the local weather forecast for tonight.	You have $50 to spend on a piece of clothing for yourself. Use the JC Penney app to find something that you might like.
You want to get some dessert and a drink late after a movie. Find a place that serves good desserts and that is open after 10 p.m.	You want to buy some pasta, diced tomatoes, and ice cream. Use the Coles app to create a list that contains all those items.
Your friend wants to watch a movie on TV tonight after 8 p.m. Find a listing of tonight's TV program and identify a movie that she may want to watch.	Using the app AA Stocks, find the current stock value of China Mobile. How did the stock change during the past month?
Find a *Tom and Jerry* video cartoon.	Use the app Flipboard for the iPad to check the latest news. Set up the app to show the news topics that interest you.
It's 6 p.m. and you need to get from West Kensington to Tufnell Park. You decide to take the underground. Find out the best way to get there, changing as few lines as possible.	You want to take a photograph of the Golden Gate Bridge from the vista point. Use the app LightTrac to find the direction of the sun's rays tomorrow at noon.

Budiu, R. & Nielsen, J. (2012). *Mobile Usability.* New Riders.

Benutzertests (auch für *User Studies*)

- Grundsätzliches Vorgehen (Dokumentieren im Testplan):
  1. Festlegen, was man genau testen will
     - Fragestellung, Ziele, Nebenbedingungen (z.B. Schulung notwendig?)
  2. Testszenarios definieren
  3. Zeitplan erstellen und Ort auswählen
     - Setup definieren
  4. Benutzer auswählen
     - Anzahl, Zusammensetzung (z.B. nach Alter, Geschlecht, Vorwissen,...)
  5. Fragebogen erstellen
     - Definieren, was man die Benutzer vor- bzw nach dem Test fragen will
  6. Aufzuzeichnende Daten definieren
     - Programm-in- und extern (z.B. zusätzliche Videoaufzeichnung,...)
  7. Aufgabenverteilung
     - Wer stellt Fragen, wer zeichnet Daten auf,...

Welche Daten sammeln?

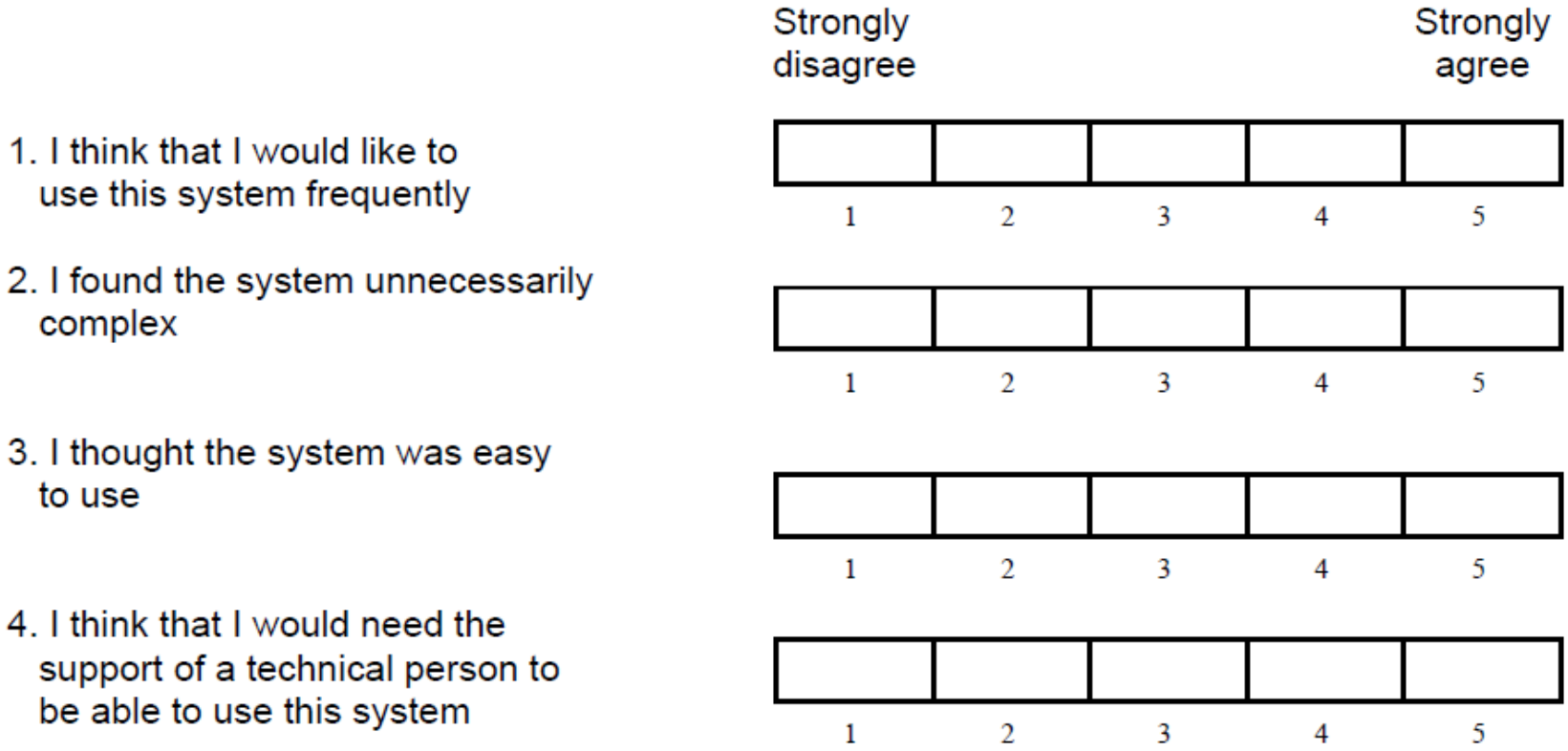
- Prinzipiell fallen zwei Arten von Daten während eines Benutzertests an:
  - Qualitative Daten
    - Beobachtungsdaten, was die Benutzer während des Tests gemacht, gesagt oder gedacht haben
  - Quantitative Daten
    - Messbare Daten, z.B. Zeiten, Fehler, Erfolge, Bewegungen

Qualitative Daten: Thinking Aloud-Methode

- Oft will man nicht nur die Resultate sehen, sondern auch wissen, was die Benutzer denken
  - => Frage die Benutzer danach
- Grundsätzliches Vorgehen: Thinking Aloud-Methode
  - Weise die Benutzer an, laut zu „denken“ bei der Ausführung des Tests
    - Erzähle, was Du denkst
    - Erzähle, was Du gerade machst
    - Welche Fragen kommen Dir in den Sinn beim Durchführen der Aufgabe?
    - Erzähle, was Du liest
  - Aufzeichnung der Gedanken
  - Manchmal muss man die Benutzer erneut motivieren, mehr zu erzählen
  - Wichtig: Nicht weiterhelfen und eingreifen, wenn der Benutzer Probleme hat

Qualitative Daten (?): Standardisierte Fragebögen

- Definiere Fragen und Antworten vor
- Für Usability-Tests gibt es auch schon einige Standardfragebögen
- Beispiel *System Usability Scale* (SUS):



Brooke, J. (1996). *SUS: a „quick and dirty" usability scale.* In P.W.Jordan, B. Thomas, B.A. Weerdmeester, and I.L. McClelland (Eds.) Usability Evaluation in Industry (189-194). London: Taylor and Francis.

Quantitative Daten

- Daten aus dem Programm selbst
  - Aufzeichnen von Mausbewegungen, Tastendrücken, Zeiten, Fehler...
  - Einige Daten nicht unbedingt eindeutig (z.B. Was bedeutet „erfolgreiche Absolvierung der Aufgabe“? Vorher definieren!)
- Aufzeichnen externer Daten
  - Video, Audio, Eye-Tracking, ...

Image: http://www.90percentofeverything.com/

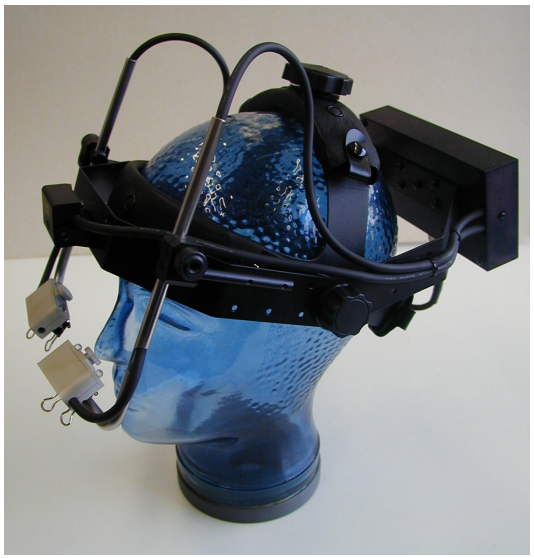
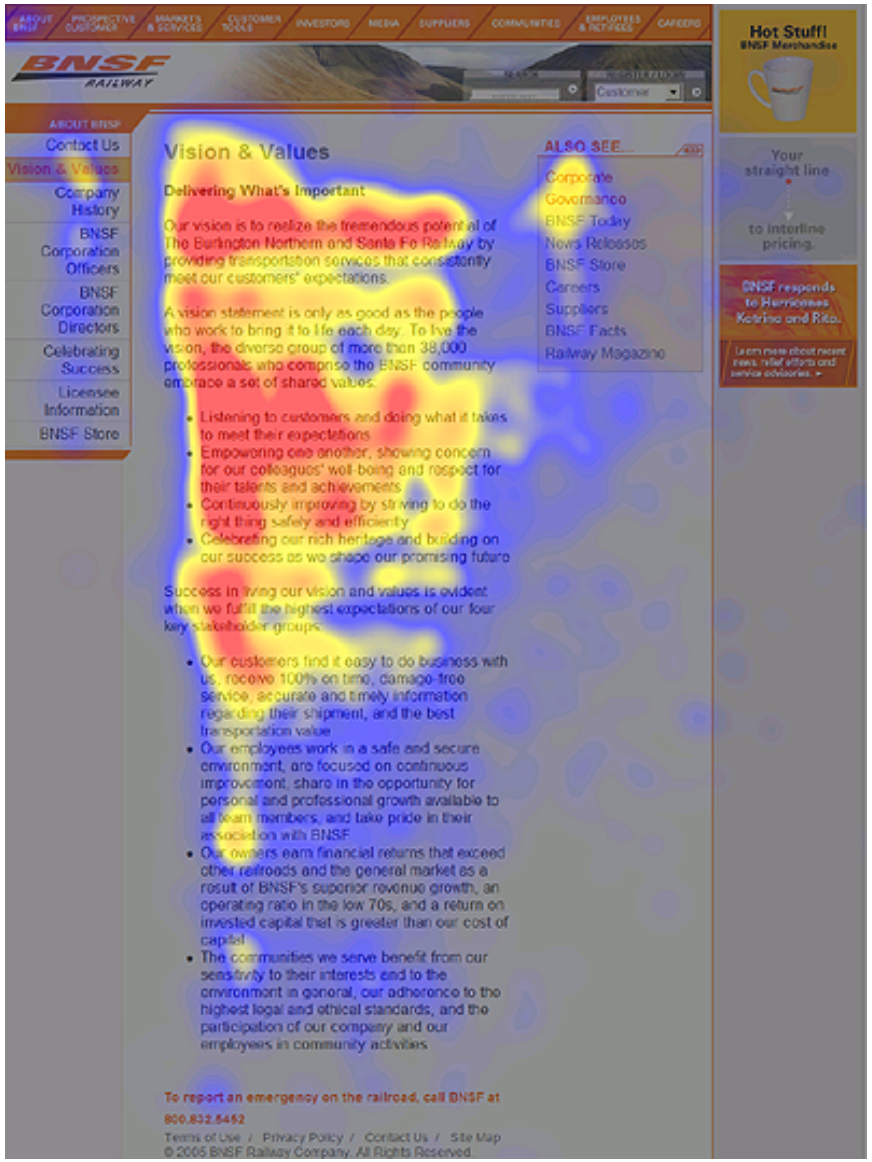


Image: http://ni.www.techfak.uni-bielefeld.de

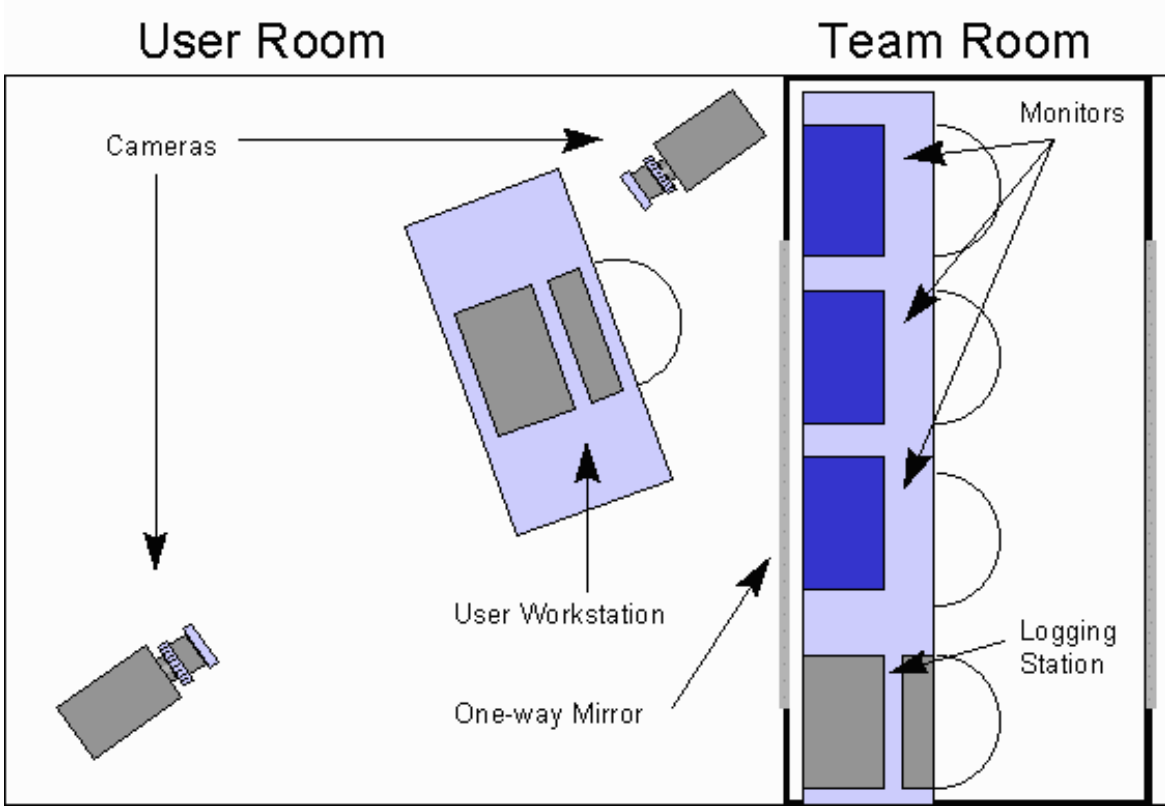
Beispiel Eye-Tracking-Ergebnisse



Heat map

Nielsen, J., Pernice, K. (2009). *Eyetracking Web Usability.* New Riders Press

Der heilige Gral: Usability Labs



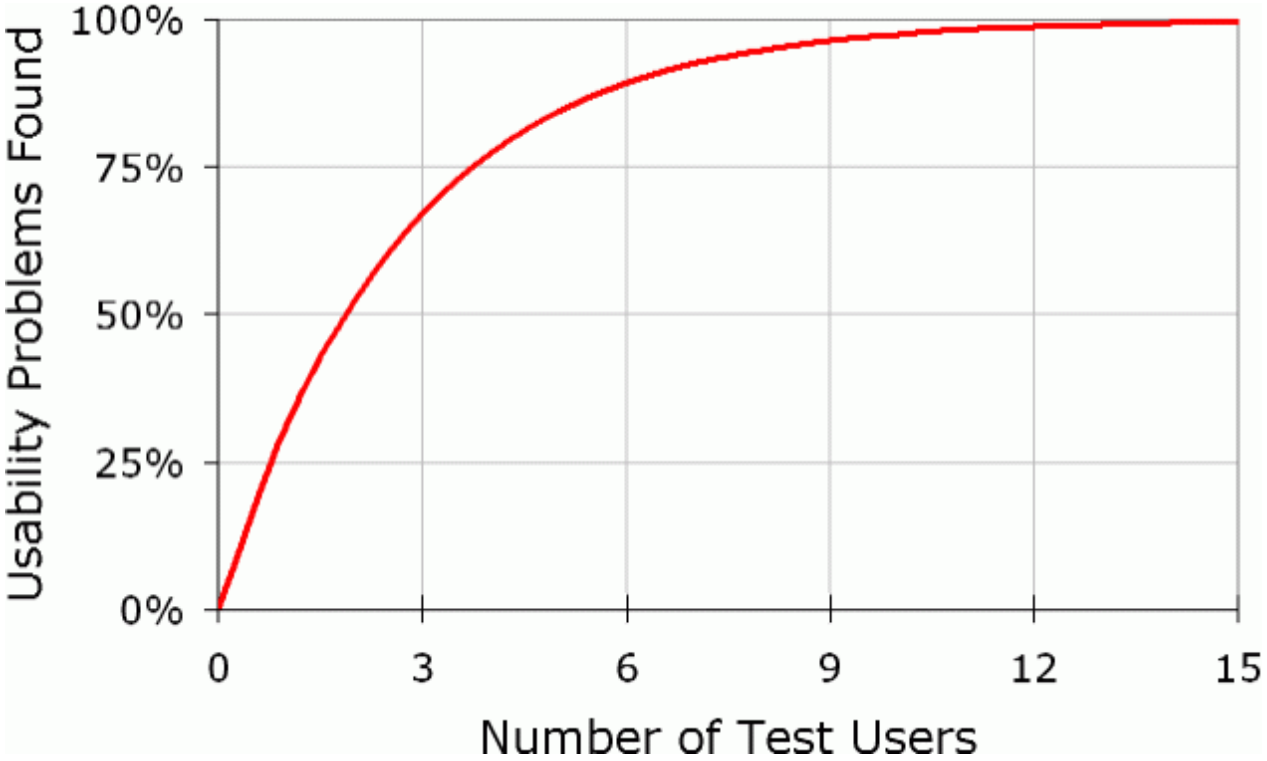
Reeves, et al. (1999). *Evaluation Report for "Remote Sensing Using Satellites"*. Available at: http://treeves.coe.uga.edu/RSUSeval/

Qualitativ oder Quantitativ?
Thinking-Aloud oder standardisiert?

- Quantitative Daten sagen nur aus, dass etwas falsch ist, aber nicht was und wie man es behebt
- Um statistisch signifikante Ergebnisse für quantitative Daten zu bekommen benötigt man sehr viele Daten
  - Das gilt auch für standardisierte Fragebögen
- Thinking-Aloud und quantitative Daten sollten nicht gleichzeitig erhoben werden
  - Zeiten werden durch das „laute“ Denken verfälscht
- Antworten aus der Thinking-Aloud-Methode sind schwer auszuwerten, wenn man zu viele Benutzer befragt
- Thinking-Aloud liefert nicht unbedingt gute Ergebnisse
  - Benutzer sagen auch gerne, was sie meinen was man hören will

Optimale Anzahl Benutzer

- Thinking-Aloud-Methode: Nielsen‘s Graph (2000)



Question still under discussion. See Barnum et a.l (2003). *The "Magic Number 5": Is It Enough for Web Testing?. CHI 2003*

- Standartisierte Fragebögen: So viele, wie das Budget hergibt

Experten vs Benutzertests

Pro Benutzertest	Pro Experten
Letztendlich kann man die Usability erst bewerten, wenn das Produkt wirklich verwendet wurde	Schnell
Benutzer sind Experten für „ihre“ Tasks	Verhältnismäßig günstig
Benutzer kennen Handlungsabläufe und das Umfeld	Benutzer wissen nicht unbedingt, was gut ist
Experten wissen manchmal „zu viel“	Kennen Standards und Normen (die manchmal nicht ganz sinnlos sind)
	Benutzer gewöhnen sich schnell an schlechte Interfaces oder suche Fehler bei sich selbst

Und wie soll man jetzt testen?

- Wie so oft gilt:

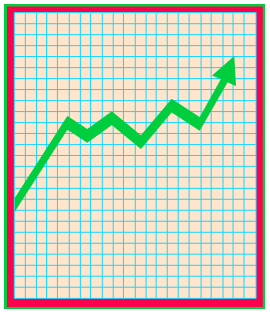
- Optimalerweise: Kombination aus allen Verfahren
  - Früher Entwicklungsstand: Test mit Thinking-Aloud-Methode
    - Dadurch können früh gravierende Design-Fehler identifiziert werden
  - Früher Entwicklungsstand: Heuristische Evaluation durch Experten
    - Überprüfung der Einhaltung von Standards
  - Spätere Phasen: Cognitive Walkthrough durch Experten
    - Liefert direkt Lösungen zu Fehlern
  - Kurz vor Fertigstellung: Vor-Test mit Thinking-Aloud/ Dann große Benutzerstudie mit mehr Teilnehmern und standardisierten Fragebögen

Auswertung

- Experten/Thinking-Aloud:
  - Liste der Anmerkungen machen (positive wie negative)
  - Eventuell ordnen nach
    - Schwierigkeit den Fehler zu Fixen
    - Schweregrad des Fehler
  - Herausfinden, warum die Schwierigkeiten/Fehler aufgetreten sind
  - Fixen der Fehler
- Benutzerstudien:
  - Statistische Analyse
  - Fixen der Fehler

Quantitative Auswertung – Ein Beispiel

- Ziel: Aufgabe soll in weniger als 30 min erledigt sein
  - 6 Benutzer wurden getestet mit Zeiten: 20, 15, 40, 90, 10, 5 min
  - Mittelwert: 30 min
  - Median: 17.5 min
- Also alles prima?
- Tatsächlich wissen wir nicht viel:
  - Die Anzahl an Benutzer ist sehr klein (n=6)
  - Die Zeiten variieren sehr stark (5-90 min)



Quantitative Auswertung – Ein Beispiel

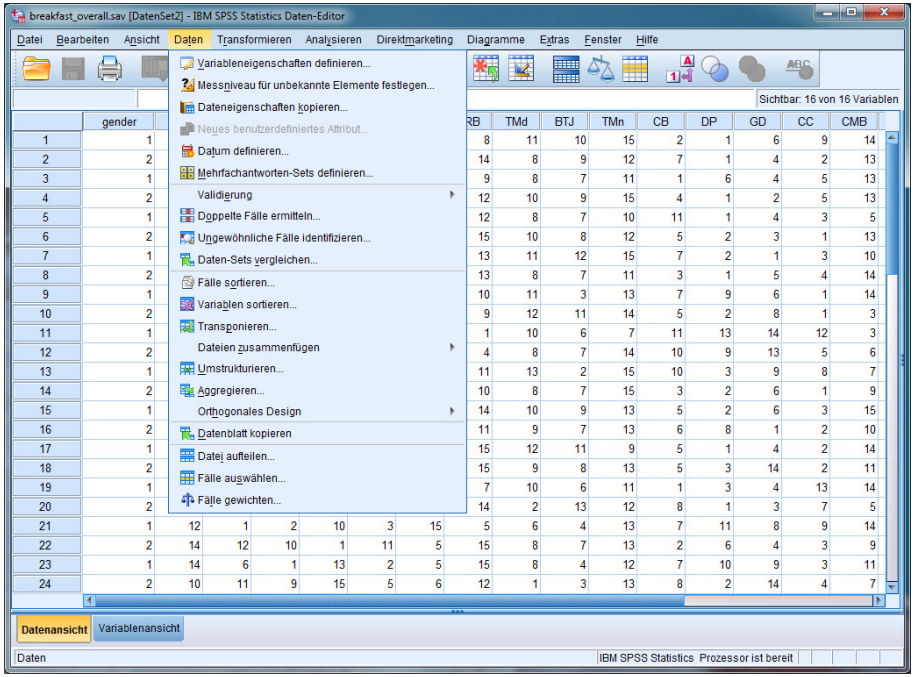
- 95% der Werte liegen in einem Intervall von 5-55 Minuten

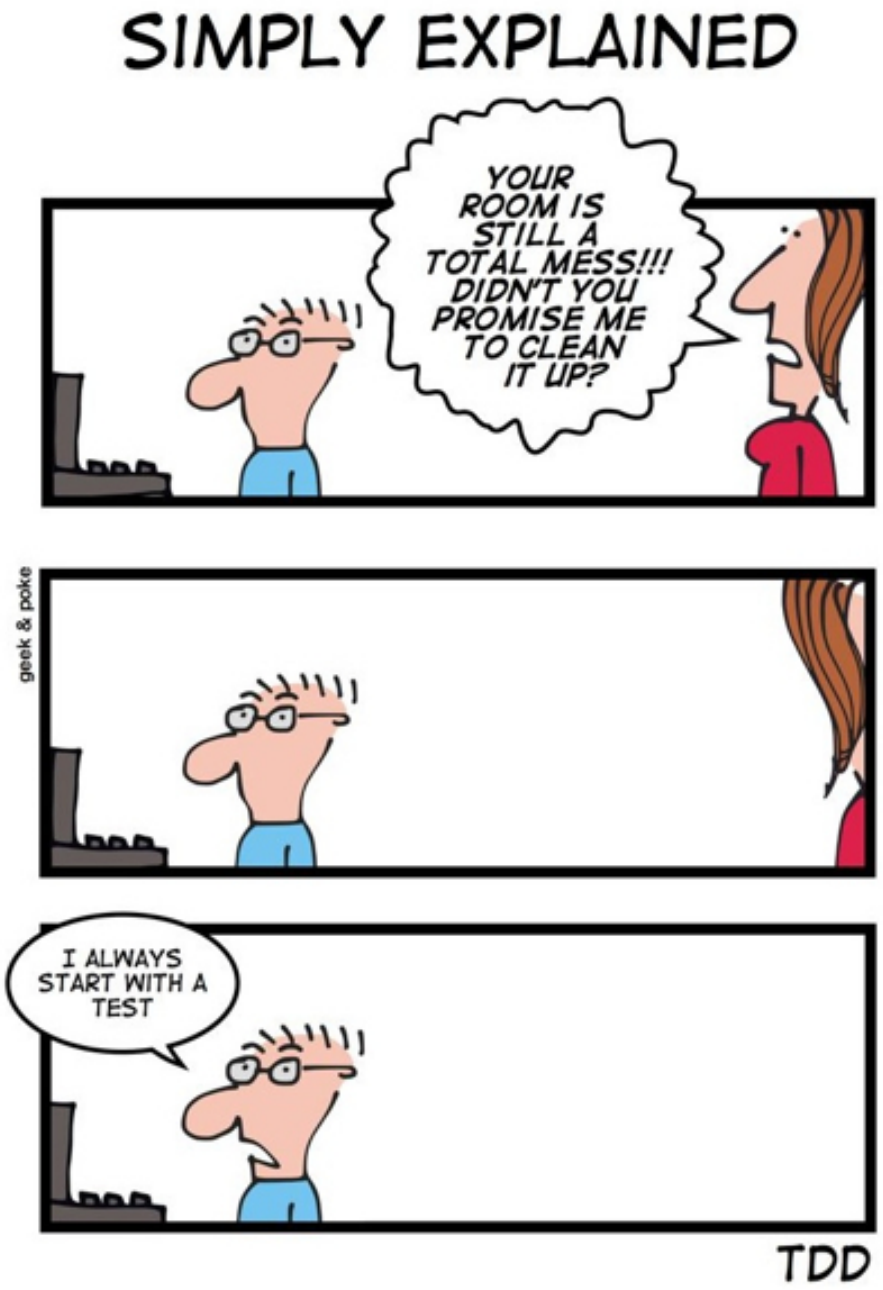
Web Usability Test Results					
Participant #		Time (minutes)			
1		20			
2		15			
3		40			
4		90			
5		10			
6		5			
	number of participants	6			
	mean	30.0			
	median	17.5			
	std dev	31.8			
	standard error of the mean	= stddev / sqrt (#samples)		13.0	
	typical values will be mean +/- 2*standard error		--> 4 to 56!		
	what is plausible? = confidence (alpha=5%, stddev, sample size)	25.4	--> 95% confident between 5 & 56		

- Im Allgemeinen sind Daten aus Benutzerstudien recht variabel
  - Man benötigt sehr viele Tests für gute Werte
  - Anzahl der Tests hängt quadratisch von der Qualität der Schranke ab

Quantitative Auswertung

- Grundwissen in Statistik ist wichtig für die Auswertung!
  - Gefühlte Auswertung führt zwangsläufig zu Fehlern
  - Oft verwendete Tests: Paired-Sample T-Test, Anova, X^2
- Spezielle Programme helfen bei der Auswertung
  - Beispiel: SPSS, R




SIMPLY EXPLAINED
YOUR ROOM IS STILL A TOTAL MESS!!! DIDN'T YOU PROMISE ME TO CLEAN IT UP?
geek & poke
I ALWAYS START WITH A TEST
TDD