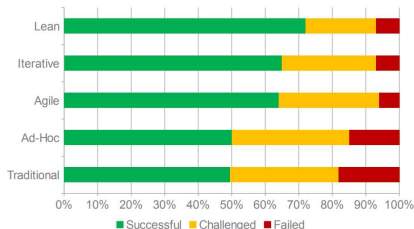
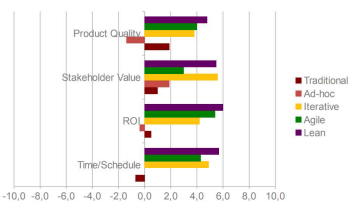
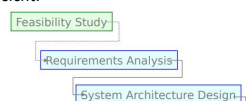
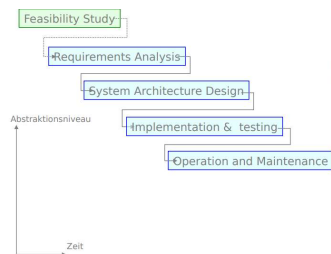


Softwarequalität

Sonntag, 12. Februar 2023 08:14

Chaos Report	<ul style="list-style-type: none">•IT-Projekte in 3 Gruppen eingeteilt<ul style="list-style-type: none">->Typ 1: Successful (erfolgreich abgeschlossen)<ul style="list-style-type: none">-On time-On budget-All features and functions as specified->Typ 2: Challenged (abgeschlossen, aber ...)<ul style="list-style-type: none">-Over time estimate-Over budget-Fewer functions and/or features->Typ 3: Failed (gescheitert)<ul style="list-style-type: none">->Project was canceled <p>->es wurden Methoden gesucht um zu schauen woran es liegt, das es sich nicht verbessert:</p> <p>COMPARING SOFTWARE DEVELOPMENT PARADIGMS: 2013</p>  <table><caption>COMPARING SOFTWARE DEVELOPMENT PARADIGMS: 2013</caption><tr><th>Paradigm</th><th>Successful (%)</th><th>Challenged (%)</th><th>Failed (%)</th></tr><tr><td>Lean</td><td>75</td><td>20</td><td>5</td></tr><tr><td>Iterative</td><td>65</td><td>30</td><td>5</td></tr><tr><td>Agile</td><td>60</td><td>35</td><td>5</td></tr><tr><td>Ad-Hoc</td><td>50</td><td>35</td><td>15</td></tr><tr><td>Traditional</td><td>50</td><td>30</td><td>20</td></tr></table> <p>->man hat gezeigt, dass Art wie Software entwickelt wird Einfluss hat</p> <p>->worauf hat es Einfluss:</p> <ul style="list-style-type: none">-Softwarequalität-Wert den Benutzer durch Software hat-Gewinn den Benutzer hat-ob Software rechtzeitig fertig wurde <p>-> Verfahren wie oben auch hier verglichen:</p> <p>COMPARING DELIVERY PARADIGMS</p>  <table><caption>COMPARING DELIVERY PARADIGMS</caption><tr><th>Category</th><th>Traditional</th><th>Ad-hoc</th><th>Iterative</th><th>Agile</th><th>Lean</th></tr><tr><td>Product Quality</td><td>1.0</td><td>2.0</td><td>3.0</td><td>4.0</td><td>5.0</td></tr><tr><td>Stakeholder Value</td><td>1.0</td><td>2.0</td><td>3.0</td><td>4.0</td><td>5.0</td></tr><tr><td>ROI</td><td>1.0</td><td>2.0</td><td>3.0</td><td>4.0</td><td>5.0</td></tr><tr><td>Time/Schedule</td><td>1.0</td><td>2.0</td><td>3.0</td><td>4.0</td><td>5.0</td></tr></table>	Paradigm	Successful (%)	Challenged (%)	Failed (%)	Lean	75	20	5	Iterative	65	30	5	Agile	60	35	5	Ad-Hoc	50	35	15	Traditional	50	30	20	Category	Traditional	Ad-hoc	Iterative	Agile	Lean	Product Quality	1.0	2.0	3.0	4.0	5.0	Stakeholder Value	1.0	2.0	3.0	4.0	5.0	ROI	1.0	2.0	3.0	4.0	5.0	Time/Schedule	1.0	2.0	3.0	4.0	5.0
Paradigm	Successful (%)	Challenged (%)	Failed (%)																																																				
Lean	75	20	5																																																				
Iterative	65	30	5																																																				
Agile	60	35	5																																																				
Ad-Hoc	50	35	15																																																				
Traditional	50	30	20																																																				
Category	Traditional	Ad-hoc	Iterative	Agile	Lean																																																		
Product Quality	1.0	2.0	3.0	4.0	5.0																																																		
Stakeholder Value	1.0	2.0	3.0	4.0	5.0																																																		
ROI	1.0	2.0	3.0	4.0	5.0																																																		
Time/Schedule	1.0	2.0	3.0	4.0	5.0																																																		
Softwarekrise	<p>-Grund: Maschinen werden immer leistungstärker und programmieren wird komplexer</p> <p>-Zwar: Qualitätssteigerung</p> <p>-Aber: gleichzeitig Steigerung der Komplexität : nichts gewonnen</p> <p>-Kompläxität Bsp</p> <p>->Anzahl der Codezeilen (Lines of Code =LOC)</p> <p>FEHLERRATEN</p> <p>Normale Software: < 25 Fehler/1000 LOC</p> <p>gute Software: 1-2 Fehler/1000 LOC</p> <p>Handy: bis zu 600 Fehler (d.h. 3 Fehler pro 1000 loc),</p> <p>Windows95: bis zu 200.000 Fehler (d.h. 20 Fehler pro 1000 loc)</p> <p>Space Shuttle: weniger als 1 Fehler pro 10.000 Zellen.</p> <p>•Ziel der Software:</p> <p>->Zuverlässig, Benutzbar, Wartbar, Effizient</p>																																																						
Softwareprozesse	<p>•Wie entwickelt man gute Software? -> 3 Modelle folgen</p> <p>•SOFTWARE-LEBENSZYKLUS</p> <ul style="list-style-type: none">-beginnt, wenn man über Software anfängt nachzudenken-endet wenn man Software nicht mehr benutzt-man unterteilt es in Phasen, welche Eingaben und Ausgaben hat-man geht von einer abstrakten Idee zu einer festen Software <p>•„PERFEKTE ENTWICKLUNGSPROZESS“</p> <ul style="list-style-type: none">• die Spezifikation wird mathematisch korrekt transformiert<ul style="list-style-type: none">• Ist die Spezifikation denn auch korrekt und vollständig?• Ist die Spezifikation eindeutig und konsistent?• Ist die Spezifikation als Basis einer solchen Transformation geeignet: d.h. formal?• Wie erfolgt die Transformation?• Ist dieses Konzept auf große und komplexe Systeme anwendbar?• In der Praxis wird der Entwicklungsprozess zwar verbessert, aber niemals „perfekt“.<ul style="list-style-type: none">• Zertifizierung des Prozesses<ul style="list-style-type: none">• ISO 9000 – Quality Management Systems• CMM - Capability Maturity Model des SEI (US DoD)• ständiges „process improvement“																																																						
Wasserfallmodell	<p>•Übersicht:</p>  <p>Software-Entwicklung als strikter „top-down“ Prozeß mit „stepwise refinement“ und „milestones“.</p> <p>→ Anforderungen</p> <p>→ Analyse</p> <p>→ Entwurf</p>																																																						



Software-Entwicklung als strikter „top-down“ Prozeß mit „stepwise refinement“ und „milestones“.

- Anforderungen
- Analyse
- Entwurf
- Implementierung und Testen
- Inbetriebnahme und Wartung (Instandhaltung)

→ es gibt Phasen, die entsprechende Milestones haben und dann wird Entwickelt

- Aufwand: -Implementierung und Aufwand: ca.15-20%
-Rest fällt auf andere Sachen, ohne Wartung

• Probleme:

PROBLEME DES REINEN WASSERFALLMODELLS & ALTERNATIVEN

- strikt sequentieller Prozess
- spätes Testen
- spätes Feedback
an Kunden und Entwickler
- praktisch kein Risikomanagement
für Kunden und Entwickler

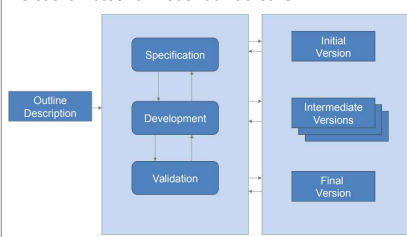


Alternative Modelle für Software-Lebenszyklen existieren in großer Zahl:
- V-Modell
- Evolutionary Development
- Spiralen-Modell
- Unified Process
- Agile Ansätze
etc.



Evolutionäre Softwareentwicklung

-versucht Wasserfallmodell aufzubrechen

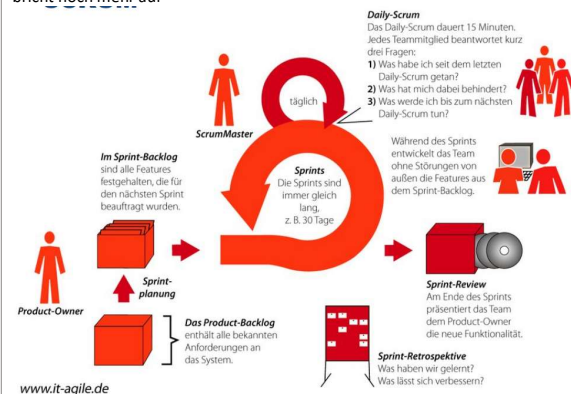


- man geht davon aus, dass man **Beschreibung** hat, von dem Problem, was man lösen will und eine Idee was am Ende raus kommen soll
- man macht also **Spezifikation** und erstellt daraus erste Version
- wenn Version einigermaßen funktioniert, dann entwickelt man eigentliche Software
→ Entwicklung in verschiedenen Phasen
→ Man hat **verschiedene Versionen** (Initial, Intermediate, Final), die Zwischendurch entstehen, anstatt Finale Version (wie in Wasserfallmodell)
- Versionen werden Kunden gezeigt um Feedback zu geben und zu bekommen
- auf Ende zu muss es **validiert** werden
- Ergebnis: Fertige Version

- PRO: -Spiegelt tatsächliche Bedürfnisse des Kunden wieder
-System kann an sich entwickelndes Problemverständnis des Kunden angepasst werden
-durch Zwischenversionen wird Kunden mehr Übersicht geboten und ihm wird genauer bewusst, was er möchte
- CONTRA: -Prozess undurchsichtig (zu viele Pfeile, wann geh ich wo hin)
-Entstehender Code häufig unstrukturiert
-Gefahr, bei Zwischenversionen „hängen zu bleiben“ und Endversion entsteht garnich

Scrum

-bricht noch mehr auf



- man hat keine großen Zwischenversionen, die dem Kunden jeden Monat präsentiert werden
→ man teilt es auf in kleine Unterabschnitte, "Sprints"
→ Abschnitte sind immer gleich lang und Zeit ist festgelegt (wird nicht anhand von Milestones festgelegt)
- am Anfang der Periode, werden Probleme festgesetzt, die man lösen möchte (ToDoListe), wichtig sind und Zeitlich schaffbar sind
→ innerhalb der Zeit werden nur diese Probleme bearbeitet
→ nach der Zeit wird ausgewertet (was hat man geschafft und was nicht und warum)
- man hat tägliche kurze Meetings (was hat man am Vortag gemacht, Review, was macht man bis morgen)
→ soll Mitglieder auf dem Laufenden halten, was im Team passiert

- wenn Kunde Sachen im Projekt ändern möchte (andere Anforderungen), dann wird dies innerhalb eines Sprints (Abs.) nicht geändert
- >Forderungen des Kunden werden erst am Ende des Abschnittes erst angenommen und bearbeitet

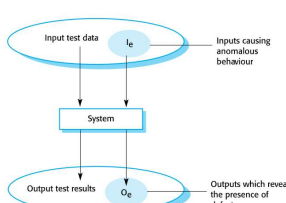
User Story

- In der agilen Softwareentwicklung eingesetzt, um Anforderungen an das System zu erfassen
- Jeweils kurze Beschreibung von etwas, das zukünftige Benutzende möchten:
 - >Als [Rolle] möchte ich [Funktionalität], damit ich [Grund]
 - >Gesammelt im Product-Backlog
- Im Sprint-Planning: Gespräch über die User Story
 - >zum genaueren Verständnis, ->zur Verfeinerung
 - >und zur Festlegung von Testkriterien (Definition of Done)
- Eigenschaften einer guten User Story: **INVEST**
 - Independent – Funktionen können unabhängig von anderen User Stories implementiert werden
 - Negotiable – Ausgestaltung kann verhandelt werden (kein „wie“)
 - Valuable – Sie nützt mindestens einem Stakeholder
 - Estimable – Aufwand zur Umsetzung kann abgeschätzt werden, Problem und Lösung klar verstanden
 - Small – Sie kann innerhalb eines Sprints umgesetzt werden
 - Testable – Product Owner kann Kriterien festlegen, mit denen überprüft werden kann, ob die Umsetzung abgeschlossen ist.
- >Wenn eine User Story immer noch zu groß ist, um in einem Sprint bearbeitet zu werden, muss sie aufgeteilt werden
- Aufteilungsansätze:-Kann man die Story so aufteilen, dass
 - >Workflowbeginn und -ende in einer Story und die Mitte in einer anderen liegt
 - >Unterschiedliche Operationen in unterschiedlichen Stories beschrieben werden
 - >Teilmenge der Geschäftsregeln in der ersten Story, weitere in weiteren Stories umgesetzt werden
 - >Nach unterschiedlichen Daten aufgeteilt wird
 - >Entlang von Schnittstellen aufgeteilt wird
 - >Mit dem Teil mit dem größten Aufwand angefangen wird
 - >Erst eine einfache Lösung und dann komplexere entwickelt werden
 - >Erst mal Basislösung und dann performante Lösung entwickelt wird

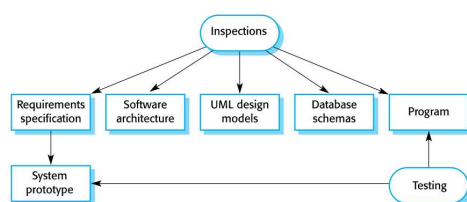
SOFTWARETESTS

- Ausführung einer Software mit künstlichen Daten um zu Prüfen
- Kann nur Fehler aufzeigen, nicht deren Abwesenheit! -> man kann nur testen ob etwas funktioniert, aber nicht ob es immer funktioniert
- Softwaretests sollten Teil eines generelleren Prozesses -> Bspw. noch statische Analysen möglich
- Ziele des Testens
 - Funktionalität überprüfen
 - >Übereinstimmung mit den Anforderungen -> geprüft ob Anforderungen erfüllt sind
 - >Für jede Anforderung gibt es mind. einen Test
 - >Ggf. Kombinationsmöglichkeiten von Features -> wenn mehrere Fkt zsm funktionieren sollen, gibt es für jede einen Test
 - Validierungstest
 - >nicht nur Spezifikation geprüft, auch ob entwickelte System am Ende die Aufgabe erfüllt
 - Fehler aufdecken
 - >Verhindern von unerwünschtem Verhalten (Falsche Berechnungen, Data Corruption, Ungewollte Interaktionen, System Abstürze, ...)
 - >Auch "falsche" Eingaben möglich
- Modellierung:

AN INPUT-OUTPUT MODEL OF PROGRAM TESTING


- White box testing: -Wissen über den inneren Aufbau einer Komponente -> ich weiß was in Kasten drin ist
 - Teilweise durch den Entwickler selbst -> man testet mit dem Wissen, interne Komponenten
- Black box testing: -man testet hier auch was in dem Kasten ist, man weiß aber nicht was drin ist, sondern nur was es tun soll
 - Keine Kenntnisse über den inneren Aufbau der Komponenten, ich bin nicht Entwickler
 - man weiß nur was Box tun soll an Hand von Spezifikation (und man testet das)
- Verifikation: -"Are we building the product right" ->The software should conform to its specification
- Validierung: -"Are we building the right product" ->The software should do what the user really requires.
- Fehlersuche:
 - >Softwareinspektion: -man schaut sich statische System/Sourcecode an und schaut ob es den Spezifikationen entspricht
 - man kann hier schon nach Fehlern suchen
 - es gibt Tools die das unterstützen, zur Dokumentation und Analyse
 - >Softwaretesting: -Concerned with exercising and observing product behaviour (dynamic verification)
 - The system is executed with test data and its operational behaviour is observed

INSPECTIONS AND TESTING

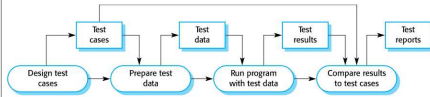


- Softwareinspektion
 - vorteilhaft: es wird keine Ausführung des Systems benötigt
 - Überprüfung einer Repräsentation des Systems durch den Menschen(Anforderungen, Design, Test Daten, ...)
 - Effektives Werkzeug der Qualitätssicherung!

- PRO: ->Systemteile können getestet werden (Das System muss noch nicht fertig sein)
- >Fehler können andere Fehler überdecken (Bei statischer Analyse spielt Fehlerinteraktion keine Rolle)
- >Kann auch noch weitere Kriterien überprüfen (Umsetzung von Standards, Wartbarkeit, ...)

- >Inspektionen und Testen ergänzen sich gegenseitig
- >Inspektionen können nur mit der Spezifikation vergleichen, nicht aber mit den echten Anforderungen
- >Inspektionen können keine nicht-funktionalen Anforderungen überprüfen (Performance, Usability, ...)

A MODEL OF THE SOFTWARE TESTING PROCESS



- man designt Testfälle -> erstellt die dann -> erstellt Daten für Testfälle -> lässt Programm mit den Daten laufen -> bekommt Resultate
- > vergleicht Resultate mit denen, die man vorher für die Testfälle hatte -> erstellt daraus Report

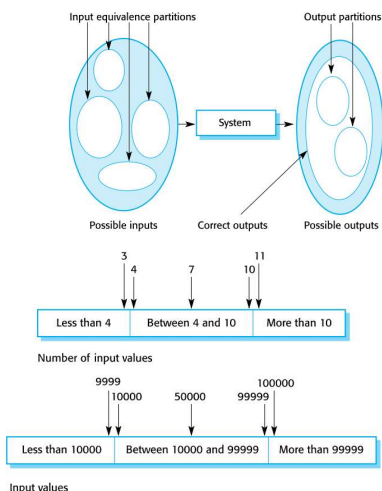
•Stufen des Testens

- >Entwickler-Tests (development tests): -Schon während der Softwareentwicklung
- im Besten Fall wird nach jeder Änderung getestet, ob alles noch läuft
- >Release-Tests: -Vollständiges System wird getestet
- Optimal durch getrenntes Test-Team, also nicht von Entwicklerteam
- >Nutzer-Tests (user tests): -Testen in eigentlicher Nutzerumgebung
- Durch Nutzer oder potentielle Nutzer getestet

Entwickler-Tests

- Unit Tests: -Testen einzelner Funktionen oder Klassen
- Funktionalität im Vordergrund
- Testen einzelner Komponenten
- Mögliche Komponenten: Einzelne Funktionen oder Methoden, Objekte mit ihren Eigenschaften und Methoden, Zusammengesetzte Komponenten, die durch Interfaces ansprechbar sind
- Komponenten Tests: -Testen des Zusammenspiels mehrerer Komponenten
- Interfaces im Vordergrund
- System Tests: -Testen des System in seiner Gesamtheit
- Zusammenspiel der Komponenten im Vordergrund
- wenn Testfälle erstellen, dann sollten 2 Gruppen abgedeckt sein
- Korrekte Funktion nachweisen (Ergeben sich aus der Spezifikation) -> welche Spezifikation erfüllen
- >man ruft Fkt mit entsprechenden Eingaben auf und schaut das Ausgaben stimmen
- Mögliche Fehler aufdecken
- >Basieren oft auf vergangenen Tests: "Was ging schief?"
- >Unerwartete Eingaben: System darf nicht abstürzen (bsp: Parameter liegt nicht im erwarteten Wertebereich, System darf nicht Abstürzen, muss Fehlermeldung liefern)
- Teststrategien:
- >Partition Tests: -Eingabe-Klassen mit gleichen Eigenschaften identifizieren -> Äquivalenzklassen
- Testdaten aus jeder dieser Klassen wählen
- >Guideline-based Tests: -Best practices
- Basierend auf Erfahrungen: "Wo passieren oft Fehler?"

EQUIVALENCE PARTITIONING



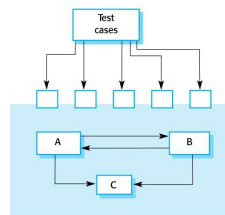
- Testing Guidelines (Listen)
- Liste ohne Elemente
- Liste mit nur einem Element
- Listen mit verschiedenen Anzahlen an Elementen
- Test, so dass das erste, mittlere oder letzte Element genutzt werden muss
- Testing Guidelines
- Jede Fehlermeldung testen
- Bei Größenbeschränkungen: -Längere Eingaben testen
- Wiederholen des gleichen Tests • Zu große oder zu kleine Ausgaben provozieren
- Ungültige Ausgaben provozieren
- Komponenten Testing
- Testen des Zusammenspiels mehrerer Komponenten
- Annahme: Einzelne Komponenten funktionieren wie erwartet

- Zusammengesetzte Komponenten: -Kombinieren verschiedener Funktionen/Methoden
 - Beispiel Wetterstation: -Abfrage eines Sensors und Darstellung des Ergebnisses
- Testen der entsprechenden Interfaces: -Übereinstimmung mit der Spezifikation

•Interface-Testing

- Aufspüren von fehlerhaften Interfaces bzw. fehlerhaften Annahmen über Interfaces
- Interface Typen:
 - Parameter Interfaces: Funktions-/Methodenaufrufe
 - Shared Memory Interfaces: Geteilte Speicherbereiche
 - Procedural Interfaces: Zusammenfassen mehrerer Methoden
 - Message Passing Interfaces: Aufruf von (Web)-Services

INTERFACE TESTING



- >Errors:
 - Falsche Nutzung eines Interfaces (Bspw. falsche Reihenfolge der Parameter)
 - Falsche Annahmen über ein Interface (Bspw. Verwenden einer Suchmethode für binäre Suchbäume auf einem unsortierten Array)
 - Timing errors (Bspw. veraltete Daten erhalten , Insbesondere in parallelen Systemen)

->Interface-Testing-Guidelines:

- Parameter an den Grenzen ihrer Wertebereiche
- Bei Pointern: Auch Tests mit Null-Pointern
- Fehlerfälle provozieren
- Stress-Tests für Systeme mit Message-Parsing
- Shared Memory Interfaces: -Reihenfolge der Operationen variieren

•System-Testing:

- Während der Entwicklung schon die Funktionen des vollständig integrierten Systems testen
- Zusammenspiel der Komponenten testen
- Überprüfen, dass Komponenten kompatibel sind, korrekt zusammenspielen und die richtigen Daten zur richtigen Zeit übertragen.
- Auch Einbindung externer Komponenten getestet -> Auch durch andere Entwickler(-gruppen) der gleichen Firma

•Use-Case-Testing:

- Use Cases aus der Spezifikation nutzen
- Use Cases modellieren durch Sequenz-Diagramme
- Meist mehrere Komponenten involviert -> Man muss also das Zusammenspiel testen
- modelliert durch DATA SEQUENCE CHART

•Test Cases aus Sequenz-Diagramm

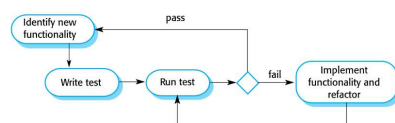
- >Ein Request wird durch ein Acknowledgement bestätigt
- >Ein Request resultiert in einem Report
- >Hier auch das Format des Report überprüfen!
- >Fehlerfälle hier ausgelassen
- >Müssen aber auch abgedeckt sein!

TDD

•Test Driven Development

- Enge Verbindung von Entwickeln und Testen
- Zuerst Tests schreiben, danach Funktionalität entwickeln
- Hauptkriterium: Erfolgreiches Bestehen der Tests
- Inkrementelle Entwicklung -> Erst wenn alle Tests erfolgreich sind, wird das nächste Feature bearbeitet
- Eingeführt v.a. durch agile Methoden wie bspw. Extreme Programming -> Kann aber auch in anderen Ansätzen eingesetzt werden

•Prozess



- Neue Funktionalität beschreiben -> Sollte eher feingranular sein
- Tests entwickeln
- Ausführen des Tests -> Sollte beim ersten Ausführen fehlschlagen
 - > Was wenn doch erfolgreich?
- Neue Funktionalität implementieren und Tests wieder ausführen
- Sobald alle Tests erfolgreich sind, nächste Funktionalität bearbeiten

•Vorteile

- Test Coverage -> Jedes Code Fragment wird durch mind. einen Test überprüft
- Regression tests
- Simplified debugging -> Wenn ein Test fehlschlägt, sollte klar sein, wo das Problem ist
 - > Neuen Code überprüfen
- System documentation -> Tests sind auch eine Art von Dokumentierung

•Regression Tests

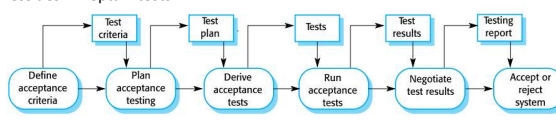
- Überprüfen, ob der neue Code nichts "kaputt" gemacht hat
- Im manuellen Testen sehr aufwendig - automatisiert aber einfach: ->Bei jeder Änderung alle Tests ausführen
- Alle Tests müssen erfolgreich sein, bevor eine Änderungen übernommen wird

Release Testing

•Allgemein

- Überprüfen einer Softwareversion, die auch außerhalb des Entwicklungsteams genutzt werden soll
- Hauptzweck: Nachweis des "Fit-for-Use" -> Funktionalität wird erbracht
 - > Performancegrenzen eingehalten
 - > Stabilität gesichert
 - > ...
- Meist Black-Box-Tests

•Release-Testing vs System-Testing

	<ul style="list-style-type: none"> -Unterschiede: -> Release Testen durch ein gesondertes Test-Team -> Ziele: -System testing: Finden von Fehlern -Release testing: Nachweis der Funktionalität <ul style="list-style-type: none"> • Requirements based Testing <ul style="list-style-type: none"> -Umsetzen aller Anforderungen in entsprechende Testfälle -> Anforderungen müssen sich auch in Tests umsetzen lassen! - Beispiel "Mentcare system" • Performance Testing <ul style="list-style-type: none"> -Testen der Performance und Reliability eines Systems. -Testumgebung sollte der des Nutzers gleichen. -Meist Sequenz von Tests, die stetig die Systemlast vergrößern. -Stress-Tests: Versuch das System zu überlasten.
User Testing	<ul style="list-style-type: none"> • Allgemein <ul style="list-style-type: none"> -Durchgeführt von Nutzen bzw. Käufern -> Testen mit üblichem Workload -Trotz System und Release Tests noch nötig -> Umgebung des Nutzers -> Erwartungen des Nutzers ev. nicht vollständig/richtig spezifiziert -> ... • Typen: -> Alpha Tests: -Testen einer ersten (unfertige) Versionen der Software <ul style="list-style-type: none"> -Oft innerhalb der Entwicklungsumgebung -> Beta Tests: -Testen einer fertigen (?) Version der Software <ul style="list-style-type: none"> -Nutzer können experimentieren und melden Probleme -> Akzeptanz Tests: -Der Käufer entscheidet, ob er das System als fertig ansieht. <ul style="list-style-type: none"> -Meist auf den Systemen des Käufers. <p>-> Prozess des Akzeptanztests</p>  <ul style="list-style-type: none"> -Definieren der Kriterien -> Sollten frühestmöglich festgelegt werden -Planung der Tests -> Zeitplan und benötigte Ressourcen festlegen -Erstellen der Tests -> Möglichst vollständige Abdeckung der Kriterien -Ausführen der Tests -> Auswertung der Testergebnisse -Entscheidung, ob Kriterien erfüllt wurden -> Annahme/Ablehnung des Systems
Zsmfassung	<ul style="list-style-type: none"> • Testen zeigt nur die Anwesenheit von Fehlern, nicht ihre Abwesenheit. • Entwicklertests sollten durch die Entwickler selbst durchgeführt werden. • Vor der Auslieferung sollte ein Test von einem anderen Team durchgeführt werden. • Entwicklertests bestehen aus Unit Tests, die einzelne Komponenten testen, Komponenten Tests, die Gruppen von Komponenten testen, und System Tests, die das Zusammenspiel des ganzen Systems oder Teilen davon testen. • Beim Testen sollte man explizit versuchen, Fehler zu finden. • Tests sollten möglichst automatisiert werden, damit sie bei jeder Änderung einfach wiederholt werden können. • TDD ist ein Verfahren, bei dem die Tests vor der Implementierung entstehen. • Akzeptanztests überprüfen, ob ein System "gut genug" ist und produktiv eingesetzt werden kann.