



Testspezifikation

SEP: Entwicklung von Projekt „CFX 100“

Version 1.0

Testspezifikation der Gruppe „sepis“ zum Software
Entwicklungsprojekt im Sommersemester 2007

Roman Moor

24.04.2007



Inhaltsverzeichnis

1. Zu testende Programmteile.....	3
2. Beschreibung.....	3
3. Zeitplan.....	3
4. Personal.....	4
5. Systemumgebung.....	4
6. Fehlerreport.....	4
6.1 Meldesystem.....	4
6.2 Bugreporting.....	4
6.3 Status.....	7
7. Testmethoden.....	7
7.1 Modultest.....	7
7.2 Systemtest.....	8
7.3 White-Box-Test.....	8
7.4 Black-Box-Test.....	9
7.5 Regressionstest.....	10
7.6 Smoke Test.....	10

1. Zu testende Programmteile

Zu testende Programmkomponenten sind:

- Datenimport der XML-Dateien
- Speicherung der Anfragen
- Produktkonfigurationskomponenten
- PDF-Export
- Preisberechnung
- Benutzereinstellungen
- Benutzeroberfläche

2. Beschreibung

Alle Programmteile werden von einer einzigen Nutzerrolle bedient, somit entfallen Nutzerrollen bei der Betrachtung der Testfälle. D.h. es erfolgt keine Unterscheidung verschiedener Nutzerrollen wie z.B. Admin, Hauptbenutzer oder Benutzer. Die Testfälle werden als Positivtests ausgelegt und dienen der Verifikation des Programms. Dabei ist festzuhalten, ob die Tests erfolgreich waren oder nicht. Beim Testen ist festzuhalten, welche Eingabe- und Ausgabe-Bedingungen galten.

Das spezifizierte Programmverhalten wird geprüft durch:

- korrekte Eingaben
- falsche Eingaben
- undefinierter Eingaben
- Fehlbedienung
- Funktionstests der einzelnen Programmteile

Die einzelnen Testfälle werden im zweiten Zyklus beim Softwaredesign festgelegt.

3. Zeitplan

Die Testphase erstreckt sich über den zweiten und dritten Zyklus. Im zweiten Zyklus werden die Testfälle anhand des Design und der Anforderungen definiert

und eine Teststrategie ausgearbeitet. Desweiteren werden in dieser Phase bereits Komponententests durchgeführt. Im dritten Zyklus findet dann der Regressions-, der Smoke-, der System- und gegebenenfalls der Performancetest statt.

4. Personal

Testleiter, Testdesigner, Tester: Roman Moor (1)*

Testdesigner, Tester: Konstantin Kramer (2/3)*

Tester: Rolf Schneider (1/3)*

In den Klammern befindet sich der geschätzte Arbeitsaufwand, wobei die 1 für 100% Auslastung steht.

5. Systemumgebung

Die Tests müssen auf aktuellen Systemen mit Windows XP Service Pack 2 durchgeführt werden. JRE 1.5 und ein PDF-Viewer (Adobe Acrobat Reader 7.0, optional auch 8.0) müssen installiert sein.

6. Fehlerreport

6.1 Meldesystem

Die komplette Fehlerverwaltung wird über das Trac-Ticketsystem abgewickelt. Diese ist unter folgender URL erreichbar ist: <http://kawg.ath.cx/sep>

Die Fehler werden von den Testern über Trac gemeldet und den jeweiligen Entwicklern zugeteilt.

Die Tickets können wie folgt bearbeitet werden:

- annehmen
- Status ändern
- anderem Entwickler zuteilen

6.2 Bugreporting

Das Bugreporting erfolgt nach folgendem Schema:

Username:

Name des Testers

Short Summary:

Kurze Beschreibung des Problems

Full Description:

Vollständige Beschreibung des Bugs. (Ist-Verhalten, Soll-Verhalten, Reproduktion, Anmerkungen)

Ist-Verhalten: Das aktuelle Verhalten des Programmteiles/der Software.

Soll-Verhalten: Das gewünschte Verhalten des Programmteiles/der Software.

Reproduktion: Anleitung wie der Entwickler den Fehler reproduzieren kann.

Anmerkungen: Optionale Anmerkung die an den Entwickler gerichtet ist.

Beispiel:

Ist-Verhalten:

Beim drücken des Buttons „PDF erzeugen“ und nach Eingabe des Zielpfads wird die erzeugte PDF in das Standardverzeichnis abgelegt.

Soll-Verhalten:

Beim drücken des Buttons „PDF erzeugen“ und nach Eingabe des Zielpfads soll die erzeugte PDF in den Zielordner abgelegt werden.

Reproduktion:

Erstelle zuerst eine neue Anfrage. Konfiguriere die Anfrage. Drücke „PDF erzeugen“.

(Bei komplizierteren Abläufen soll detaillierter beschrieben werden.)

Component:

Betreffende Komponente wählen mit dem Präfix „BUG-“.



Möglichkeiten: BUG-XML

BUG-SAVE

BUG-VERIFIKATION

BUG-PDF

BUG-CALC

BUG-USER

Version:

Die Version des getesteten Programmmoduls.

Severity:

Beschreibt den Effekt den der Bug auslöst.

Möglichkeiten: Blocker – Systemabsturz / -blockade

Critical – Kritisch / Anforderung nicht erfüllt

Major – Schwerwiegend / Funktion gestört

Normal – Funktion eingeschränkt

Minor – Kleine Fehlfunktion

Trivial – Kleiner Fehler ohne Auswirkung

Beispiele: Trivial – Falsches Datum wird angezeigt.

Minor – PDF wird in das falsche Verzeichnis abgelegt.

Critical – Komplett fehlende Funktion.

Keywords:

Schlagwörter zur besseren Auffindung des Bugs.

Priority:

Dringlichkeitsstufe des Bugs.

Möglichkeiten: Highest (Schnellstmöglich Beseitigung)



High (Schnelle Beseitigung)

Normal

Low (Beseitigung wenn Zeit vorhanden)

Lowest (Beseitigung wenn alles erledigt ist)

Milestone:

Angabe bis welchen Meilenstein der Bug gefixt sein sollte.

Assign to:

Welchem Entwickler die Aufgabe zugeteilt wurde.

6.3 Status

Die Tickets können folgende Zustände haben:

FIXED	- Fehler behoben
INVALID	- Fehler noch nicht behoben (offen)
WONTFIX	- Wird nicht behoben
DUPLICATE	- Doppelmeldung (vom anderen Tester)
WORKSFORME	- Nach Reproduktion kein Fehler

7. Testmethoden

Die von uns vorgesehenen Testmethoden werden im Folgenden näher beschrieben.

7.1 Modultest

Der Modultest (auch Komponententest oder engl. unit test) ist Teil eines Softwareprozesses (z. B. nach dem Vorgehensmodell des Extreme Programming). Er dient zur Verifikation der Korrektheit von Modulen einer Software, z. B. von einzelnen Klassen. Als Voraussetzung für Refactoring kommt ihm besondere Bedeutung zu. Nach jeder Änderung sollte durch Ablauf aller Testfälle nach Programmfehlern gesucht werden. Bei der testgetriebenen Entwicklung, auch

TestFirst-Programmieren genannt, werden die Modultests parallel zum eigentlichen Quelltext erstellt und gepflegt. Dies ermöglicht bei automatisierten, reproduzierbaren Modultests, die Auswirkungen von Änderungen sofort nachzuvollziehen. Der Programmierer entdeckt dadurch leichter ungewollte Nebeneffekte oder Fehler, die durch seine Änderung verursacht wurden.

Ein Komponententest ist ein ausführbares Codefragment, welches das sichtbare Verhalten einer Komponente (z. B. einer Klasse) verifiziert und dem Programmierer eine unmittelbare Rückmeldung darüber gibt, ob die Komponente das geforderte Verhalten aufweist oder nicht. Durch diese Rückmeldung wird die Wartbarkeit z. B. durch Refactoring vereinfacht oder sogar erst ermöglicht. Komponententests sind ein wesentlicher Bestandteil der Qualitätssicherung in der Softwareentwicklung.

Modultests sind eine geeignete Vorstufe zu Integrationstests, welche wiederum zum Testen mehrerer voneinander abhängiger Komponenten im Zusammenspiel geeignet sind. Im Gegensatz zu Modultests werden Integrationstests meist manuell ausgeführt.

7.2 Systemtest

Systemtests sollen die komplette Funktionalität einer Software nachweisen. Dieser besteht aus zwei Gliederungspunkte, dem funktionalen Systemtest und dem nicht funktionalen Systemtest. Der funktionale Systemtest überprüft ein System in Bezug auf funktionale Qualitätsmerkmale wie Korrektheit und Vollständigkeit. Nicht funktionale Qualitätsmerkmale, wie z.B. die Sicherheit, die Benutzbarkeit, die Interoperabilität, die Prüfung der Dokumentation oder die Zuverlässigkeit eines Systems, werden über den nicht funktionalen Systemtest einer Prüfung unterzogen.

7.3 White-Box-Test

Der Begriff White-Box-Test (auch Glass-Box-Test) bezeichnet eine Methode des Software-Tests (Testmethode), bei der die Tests mit Kenntnissen über die innere Funktionsweise des zu testenden Systems entwickelt werden. Im Gegensatz zum Black-Box-Test ist für diesen Test also ein Blick in den Quellcode gestattet, d. h. es wird am Code geprüft.

Will man ein System auch in seinen Teilsystemen testen, benötigt man dazu Kenntnisse über die innere Funktionsweise des zu testenden Systems. White-Box-Tests eignen sich besonders gut, um in Erscheinung getretene Fehler zu lokalisieren, d. h. die fehlerverursachende Komponente zu identifizieren, und als Regressionstest ein Wiederauftreten des Fehlers bereits an der Komponente zu vermeiden.

Weil die Entwickler der Tests Kenntnisse über die innere Funktionsweise des zu testenden Systems besitzen müssen, werden White-Box-Tests von dem selben Team, häufig sogar von denselben Entwicklern entwickelt wie die zu testenden Komponenten. Spezielle Testabteilungen werden für White-Box-Tests in der Regel nicht eingesetzt, da der Nutzen speziell für diese Aufgabe abgestellter Tester meist durch den Aufwand der Einarbeitung in das System eliminiert wird.

Es lässt sich sagen, dass White-Box-Tests alleine als Testmethodik nicht ausreichen. Eine sinnvolle Testreihe sollte White-Box-Tests und Black-Box-Tests kombinieren.

7.4 Black-Box-Test

Black-Box-Test bezeichnet eine Methode des Software-Tests, bei der die Tests ohne Kenntnisse über die innere Funktionsweise des zu testenden Systems entwickelt werden. Er beschränkt sich auf funktionsorientiertes Testen, d. h. für die Ermittlung der Testfälle wird nur die Spezifikation (gewünschte Wirkung), aber nicht die Implementierung des Testobjekts herangezogen. Die genaue Beschaffenheit des Programms wird nicht betrachtet, sondern vielmehr als Black Box behandelt. Nur nach Außen sichtbares Verhalten fließt in den Test ein.

Zielsetzung:

Ziel ist es, die Übereinstimmung eines Softwaresystems mit seiner Spezifikation zu überprüfen. Ausgehend von formalen oder informalen Spezifikationen werden Testfälle erarbeitet, die sicherstellen, dass der geforderte Funktionsumfang eingehalten wird. Das zu testende System wird dabei als Ganzes betrachtet, nur sein Außenverhalten wird bei der Bewertung der Testergebnisse herangezogen. Testfälle aus einer informalen Spezifikation abzuleiten, ist vergleichsweise aufwändig und je nach Präzisionsgrad der Spezifikation u. U. nicht möglich. Oft ist daher ein vollständiger Black-Box-Test ebenso wenig wirtschaftlich wie ein vollständiger White-Box-Test.

Black-Box-Tests vermeiden, dass Programmierer Tests "um ihre eigenen Fehler herum" entwickeln und somit Lücken in der Implementierung übersehen. Ein Entwickler, der Kenntnisse über die innere Funktionsweise eines Systems besitzt, könnte unabsichtlich durch gewisse zusätzliche Annahmen, die außerhalb der Spezifikation liegen, einige Dinge in den Tests vergessen oder anders als die Spezifikation sehen. Als weitere nützliche Eigenschaft eignen sich Black-Box-Tests auch als zusätzliche Stütze zum Überprüfen der Spezifikation auf Vollständigkeit,

da eine unvollständige Spezifikation häufig Fragen bei der Entwicklung der Tests aufwirft.

7.5 Regressionstest

Unter einem Regressionstest (v. lat. Regression = Rückschritt) versteht man in der Softwaretechnik die Wiederholung aller oder einer Teilmenge aller Testfälle, um Nebenwirkungen von Modifikationen in bereits getesteten Teilen der Software aufzuspüren. Solche Modifikationen entstehen regelmäßig z. B. aufgrund der Pflege, Änderung und Korrektur von Software. Der Regressionstest gehört zu den dynamischen Testtechniken.

Aufgrund des Wiederholungscharakters und der Häufigkeit dieser Wiederholungen eignen sich Regressionstests gut für eine automatisierte Ausführung.

In der Praxis steht der Begriff des Regressionstests für die reine Wiederholung von Testfällen. Die Testfälle selbst müssen anhand anderer Techniken spezifiziert und mit einem Soll-Ergebnis versehen sein, welches mit dem Ist-Ergebnis eines Testfalls verglichen wird. Ein direkter Bezug auf die Ergebnisse eines vorherigen Testdurchlaufs findet nicht statt.

Als Framework für Regressionstests eignet sich JUnit besonders gut.

7.6 Smoke Test

Smoke testing ist ein Begriff aus dem Englischen, gebräuchlich im handwerklichen Bereich (z. B. in der Klempnerei, Elektronik oder beim Bau von Holzblasinstrumenten) wie auch in der Softwareentwicklung. Es bezeichnet den ersten Probelauf nach einer Reparatur oder der ersten Implementierung eines neuen Algorithmus um sicherzugehen, dass das Gerät oder die Programmfunktion nicht schon ansatzweise fehlschlägt. Nachdem der Smoke Test zeigt dass die Rohre nicht lecken, die Stromkreise nicht durchbrennen oder das Programm nicht gleich abstürzt, kann die Apparatur eingehender geprüft werden.