Fehler in Planung und Implementierung einer Open-Test-Architecture-Schnittstelle (für das Mercury Quality Center) in Java

Ingo Mohr
Institut für Informatik
FU Berlin
24.05.2007

Inhalt

Freie Universität Berlin

- 1. Einstieg: Software-Fehler
- 2. Open Test Architecture (OTA)
- 3. Planung
- 4. Implementierung
- 5. Zusammenfassung

1. Einstieg: Software-Fehler

- Softwarefehler – auch Programmfehler, Bug, Defect
- Kommen vor in (Computer-)Programmen durch
  - Nicht berücksichtigte Zustände in der Programmlogik
  - Ungenauigkeit, Unvollständigkeit, Mehrdeutigkeit, Missverständnisse während des gesamten Entwicklungsprozesses

1. Einstieg: Software-Fehler

Freie Universität Berlin

- Historischer Einblick
  - Im 19. Jahrhundert stören “knabbernde 'Bugs'” Telefon-Übertragung (Knistern, Rauschen)
  - Grace Hopper (1945/47): “Motte im Relais des *Mark II Aiken Relay Calculator* führt zu Fehlfunktion” - Log-Eintrag: “first actual case of bug being found”

1. Einstieg: Software-Fehler

- Schätzung: Je 1000 Zeilen Quelltext: 2 Fehler
- Annahme: Jedes nicht-triviale Computerprogramm enthält Fehler
- Fehler-Häufung in neuen Projekten mit “Badewannenkurve” abgeschätzt (betrachtet auf gesamten Entwicklungsprozess)
  - Viele Fehler zu Beginn
  - Beheben einer großen Fehleranzahl führt zu Phase mit wenig Fehlern
  - Erneuter Anstieg der Fehlerkurve zum Ende der Entwicklung

1. Einstieg: Software-Fehler

- Arten von Software-Fehlern
  - **Syntaktisch** (Verstoß gegen grammatikalische Regeln in der benutzten Programmiersprache)
  - **Semantisch** (treten während der Laufzeit eines Programms auf)
    - Fehler in Implementierung
    - Speicherlecks, auch mit Garbage-Collector
  - **Design** (Probleme in der (bzw. durch die) Konzeptionierung)
  - **Regression** (Erneutes Auftauchen von in einer vorigen Programmversion behobenen Fehlern)
  - **Bedienkonzept** (Programmverhalten anders als Annahme des Anwenders)

1. Einstieg: Software-Fehler

- Vermeiden von Fehlern
  - *Leitsatz*: Zusammenspiel von frühem Auftreten und spätem Entdecken von Fehlern führt zu hohem Aufwand, diese zu beheben.
  - **Planung**: "gute und geeignete" Planung. Verwendung von Prozessmodellen
  - **Analysephase**: geeignetes Programmier-Paradigma wählen (aber: große Erfahrung nötig, sonst evtl. gegenteiliges Resultat)
  - **Designphase**: defensives Programmieren, Ausnahmebehandlung, Redundanz
  - **Implementierung**: Überwachung von Quell- und Binärcode (manuell, automatisiert)

1. Einstieg: Software-Fehler

- **Testen**: (Unit-)Tests anhand Spezifikation, Einführung von Testphasen (bspw. Alpha-, Beta)
- (in speziellen Fällen): **Beweis** der Fehlerfreiheit (Achtung: Beweis auch zumeist recht fehleranfällig)

- Reproduzierbarkeit
  - (grobe) Klassifizierung
    - “Mandelbug” (nach Benoît Mandelbrot): Fehler, die chaotisch oder nichtdeterministisch erscheinen
      - Erneutes Auftreten auch bei scheinbar gleichen äußeren Bedingungen unsicher
      - Grund: Verzögerung zwischen Auftreten des Fehlers und dem eigentlich gefundenen Problem
      - Grund: Beeinflussung des Fehlerauftretens durch äußere Elemente der Software (VM, Betriebssystem, ...) - z.B. Fehler in nebenläufigen Umgebungen durch mangelnde Sequentialisierung – führt zu Wettlaufsituation und damit problematischer Ausführungsreihenfolge von Prozessen
    - “Bohrbug” (nach Niels Bohrs Atommodell)
      - Äußert sich unter einfachen Bedingungen relativ zuverlässig und ist dementsprechend leicht aufzuspüren

2. Open Test Architecture (OTA)

Freie Universität Berlin

- Spezifikation einer Schnittstelle zum **Mercury Quality Center** (*HP Mercury* - vormals *Mercury Interactive*)
- **Mercury Quality Center (MQC)**: web-basiertes System zum Erstellen und Verwalten von (automatisierten) Qualitätstest für zahlreiche Applikationsumgebungen
- Zugriff:
  - COM Interface -> OTAClient.dll -(HTTP/DCOM)->
  - MQC-Server -> Datenbank(en)
- Spezifikation:
  - Datenzugriff (Baumstrukturen für Tests, Requirements, ...)
  - Remote-Access von Tests, Defect-Tracking, ...

3. Planung

- Ziel
  - Schaffung einer OTA-Schnittstelle in Java
  - Erstellen, Editieren, Auslesen von Requirements und Tests (bestehend aus Testsets, Tests, Runs und Steps (hierarchisch))
  - Allgemeine Nutzbarkeit in Java-basierten Systemen
- Rahmenbedingung
  - Sukzessive Entwicklung: Fertige Programmteile alleinstehend nutzbar

3. Planung

- Konzept
  - Grundzug:
    - “Ease of use”
    - Transparenz
    - Erweiterbarkeit (da nur Teilimplementierung der Schnittstelle)
    - Defensives Programmieren, Ausnahmebehandlung
    - Nutzung des JACOB Projektes als COM-Bridge (Java COM-Bridge Implementierung): wg. existenter Vorimplementierung eines Teils der zu umfassenden Aufgaben und entsprechener Erfahrungen mit JACOB
    - Performanzsteigerung durch Caching bearbeiteter Daten
    - (Implementierung) Modulare Tests zur Implementierung (Junit)
    - Kein COM-Bridge-basierter Code zur Anwendung nötig (Wrapping)

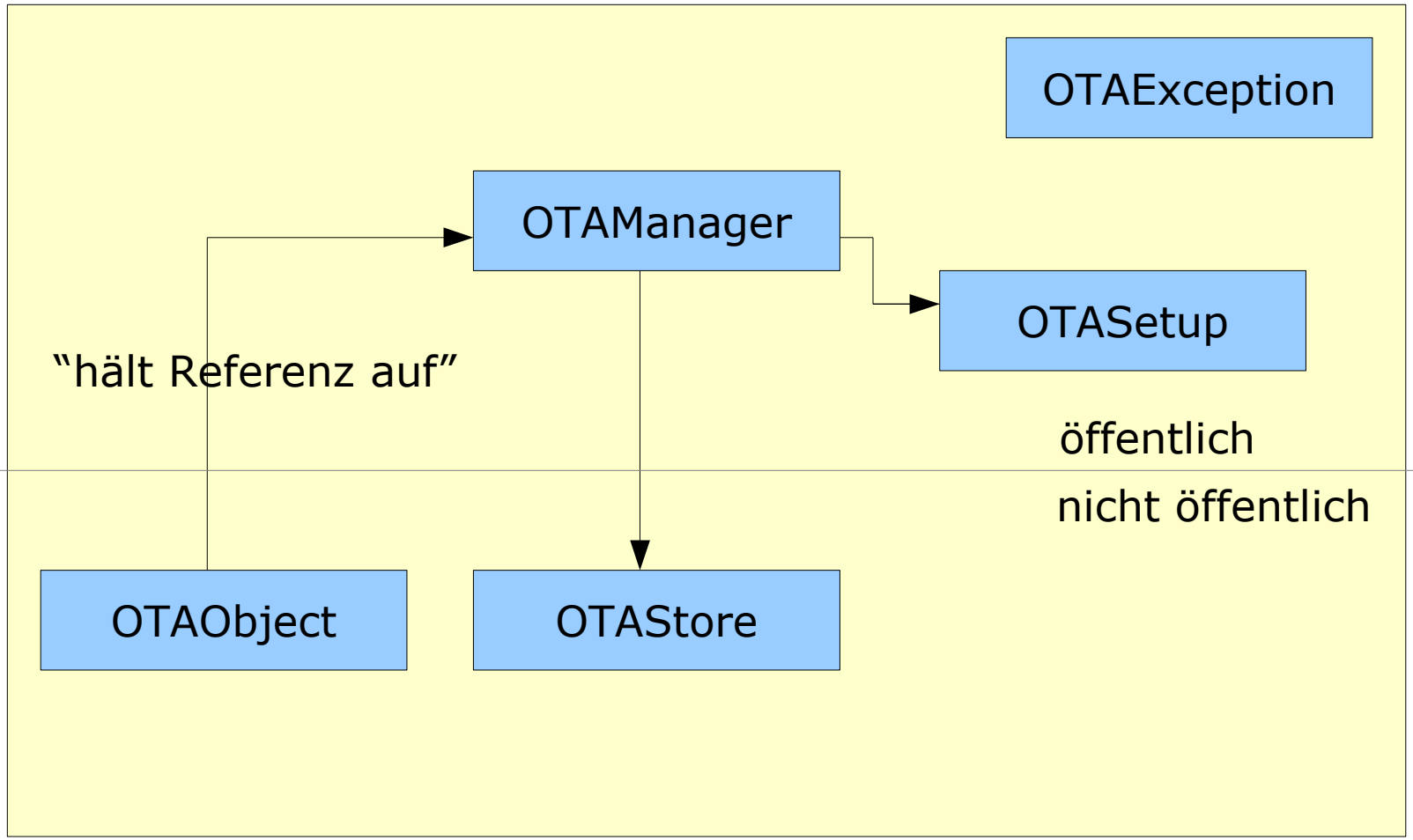
- Klassen-Konzeption:
  - Kern-Klasse OTAManager, erstellt für ein Setup (OTASetup) mit den Methoden
    - connect(): Herstellung der Verbindung
    - disconnect(): Schließen der Verbindung
    - dispose(): Freigabe aller nativen Referenzen (COM-Interface)
  - OTAManager hat Zugriff auf TestSetTreeManager (Wurzel des Testdatenbaumes im OTA-Interface)
  - OTAStore (ohne öffentlichen Zugriff) zur Verwaltung der JACOB-basierten Objekt-Referenzen (müssen disposed werden), Steuerung durch OTAManager

3. Planung

- OTAObject als Basis aller Daten-Implementierungen mit (nicht öffentlichem) Zugriff auf OTAManager und Methode dispose().
- Gängige Bezeichner für Datenzugriff (getX, setX, createX, getChildren, ...) - später geändert in 1:1-Benennung nach OTA (Vorteil: OTA-Spezifikation kann auch als “HowTo” benutzt werden, Transparenz)
- OTAException als zusammenfassender Fehlertyp für alle möglichen (prozessbedingten) Fehler (nicht für bspw. IllegalArgumentException)

Freie Universität Berlin

- Klassenkonzeption



3. Planung

- Lebenszyklus:
  - Setup **`s`** erstellen
  - OTAManager für **`s`** erstellen
  - OTAManager.connect()
  - Datenzugriff, Ermöglichung der Referenzbereinigung durch OTAObject.dispose()
  - OTAManager.disconnect()
  - OTAManager.dispose()

- Fehler:
  - **Caching**: Nicht realisierbar, weil nicht sicher gestellt werden kann, dass die Daten noch “echt” sind (weitere Zugriffe auf MQC möglich)
    - Natur: Design
    - Aufgefallen: zu Beginn der Implementierung
    - Lösung: Caching gestrichen und durch Ersatz-Teilkonzept (OTADataNode) ersetzt.
    - Aufwand: gering

- Fehler:
  - **OTAManager.dispose() am Ende des Lebenszyklus'**:
    - Natur: Design
    - Aufgetreten: während der Beta-Testphase
      - Fehlerträchtig: Anwender kann dispose-Aufruf “vergessen”
      - => Kann zu Speicherleck im Lebenszyklus des benutzenden Programms führen
    - Grund: Nichtberücksichtigung d. Zusammenhangs zwischen Disconnect und dispose
    - Lösung: disposal im disconnect vorgenommen.
    - Aufwand: gering

- Fehler:
  - **OTAObject.dispose():** gibt nur die eigene (COM-)Referenz auf
    - Natur: Design
    - Aufgetreten: noch nicht (als solcher)
      - Fehlerträchtig: Anwender kann “vergessen”, den Subbaum des Objekts zu disposen => führt zu Speicherleck im Lebenszyklus des benutzenden Systems
    - Lösung: dispose-Aufruf führt automatisch zur Freigabe aller nativen Referenzen des Subbaumes des Objekts.
    - Aufwand: mittel bis hoch (wäre gering, wenn Node-Abstraktionsschicht (wie im “Substitutions-Caching”) schon hier zu finden wäre – weiterer konzeptioneller Fehler?)

4. Implementierung

- Vorgehensweise:
  - Basisklassen
    - Klassen implementieren
    - Ungültige Parameter durch IllegalArgumentException abfangen
    - Unit Tests für alle Konstruktoren u. Methoden schreiben, testen
  - Basismodul
    - Basisklassen (BaseField, BaseFieldEx, SysTreeNode, ...) (OTA-Basis-Strukturen) auf Basis von OTAObject implementieren
    - Unit Tests..., testen
  - Datenklassen (Requirements, Test-Klassen)
    - Iterativer Vorgang:
      - Modul (Requirements, Testsets, Tests, ..., Steps) inkl. Interfaces auf Basismodul aufsetzen und implementieren
      - Unit-Tests... testen
      - Projekt als i-te Stage-Beta zum Testen freigeben

4. Implementierung

- Fehler
  - (näherungsweise n/2 bei n (COM-)Funktionalitäten)
  - Größtenteils COMFailExceptions bei hinzukommenden Funktionalitäten, weniger NullPointerExceptions (keine ArrayIndexOutOfBoundException – weil Arbeit auf Collection-Basis)
    - Natur: Semantisch
    - Aufgetreten (a): während der Unit-Tests
      - Lösung: Fehlerberichtigung im JACOB-Aufruf
      - Aufwand: insgesamt mittel bis hoch
    - Aufgetreten (b): während der (i-ten Stage-)Beta-Tests (selten)
      - Lösung: Fehlerberichtigung im JACOB-Aufruf + Erweiterung der Unit-Tests
      - Aufwand: mittel

- Weitere Fehler
  - Fehler durch unverarbeitbar formatierte Strings (unerlaubte Sonderzeichen) => führt ebenfalls zu ComFailExceptions
    - Natur: Semantisch
    - Aufgetreten: Während der Beta-Tests
    - Lösung: Vorverarbeiten (Substitution) unerlaubter Sonderzeichen
    - Aufwand: gering

4. Implementierung

- Syntaktische Fehler
  - Schwer nachzuvollziehen, da mit Eclipse incl. (default-eingestellter) Code-Analyse gearbeitet wurde:
    - Warnungen bei unbenutzten lokalen Variablen
    - Syntax-Error-Highlighting
    - Prävention durch Code-Assistance und -Completion
- Regressionsfehler
  - Keine (bisher), aber:
    - Mit 56 Klassen und Interfaces auch eher kleines Projekt
- Bedienkonzeptionelle Fehler
  - Keine (keine GUI, Zugriffe 1:1 aus OTA-Spezifikation übernommen)

- Mandelbugs
  - Verlust der Verbindung zum MQC
    - Trat sehr sporadisch auf
    - Nicht reproduzierbar innerhalb des Projekt-Quellcodes
    - Schätzungsweise zurückzuführen auf äußere Einflüsse wie bspw.
      - Verbindungs-Fehler außerhalb des lokalen Betriebssystems
      - Eventuelle Fehler in JACOB
      - Eventuelle Fehler im MQC
    - Vielleicht auch Folgefehler bei für ein externes System (JACOB, MQC) nicht verarbeitbaren Daten

- Uneigentlicher Mandelbug
  - Speicherlecks
    - Zunächst als Mandelbug missverstanden
    - Lassen sich eher schnell reproduzieren (Mit dispose()-Aufruf und Store-Referenzierungen nur zwei Verursacher-Möglichkeiten)
    - Natur: Semantisch
    - Aufgetreten: Während der Beta-Tests
    - Grund: Bei Freigabe der Referenzen wurden die OTAObject-Referenzen im OTAStore nicht freigegeben
    - Aufwand: mittel (viel grübeln, letztendlich eine Zeile Code ändern)

5. Zusammenfassung

- An diesem nur kleinen Projekt lässt sich sehr schön sehen, dass Software-Fehler praktisch “allgegenwärtig” sind.
- Die Annahme, dass jede nicht-triviale Software auch Softwarefehler enthält, sieht sich (einmal mehr) unterstrichen.
- Auch die so genannte “Badewannenkurve” der Fehlerhäufung über den Zeitraum der Entwicklung eines Projektes wird für dieses Beispielprojekt bestätigt.

Vielen Dank!

Literaturangaben

- HP Mercury, http://www.mercury.com/de/
- Artikel Programmfehler, http://de.wikipedia.org/wiki/Programmfehler
- TestDirector Open Test Architecture Guide, Version 8.0 (in "TestDirector for Quality Center")
- JACOB Project, http://danadler.com/jacob bzw. http://sourceforge.net/projects/jacob-project/