3.– 6. September 2012
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Fehlerpreväntion

Vom Umgang mit Komplexität in der Softwareentwicklung

Joachim Hofer

imbus AG

@johofer, http://jmhofer.johoop.de

Fahrplan

- Wie entstehen Fehler?
- Gegenmaßnahmen
- Komplexität: Arten, Abgrenzung
- Komplexität reduzieren
- Fazit

Kurzvorstellung

- > 10 Jahre Teamleiter Softwareentwicklung
- > 10 Jahre *zu viel* Enterprise Java
- Praktizierender™ Scrum Master
- Schwerpunkte **Test** und Code Coverage
- **Scala-Enthusiast**
- Autor von eCobertura (inzwischen veraltet...)
- Autor diverser **sbt-Plugins**
  - jacoco4sbt, findbugs4sbt, cpd4sbt, ant4sbt etc
- sporadischer Blogger (und Zwitscherer)

Wie entstehen Fehler? (1)

„Irren ist menschlich“ - tja...

- Menschliche Schwächen
- Handwerkliche Fehler
- Prozessfehler

Wie entstehen Fehler? (2)

- Menschliche Schwächen
  → Kognitionspsychologie!
  - Confirmation Bias
  - Intuitionsfehler
  - Logikfehler
  - Probleme mit Komplexität, Dynamik, Nichtlinearität

Wie entstehen Fehler? (3)

- Handwerkliche Fehler
  - → `equals()` / `hashCode()`
  - Programmieren ist schwierig!
- Prozessfehler
  - Kommunikation
  - organisatorischer Kontext
  - Theorie ↔ Praxis

Gegenmaßnahmen

- Handwerkliche Fehler
  - sein Handwerk beherrschen (!)
  - auf dem aktuellen Stand der Technik bleiben
- Prozessfehler
  - möglichst leichtgewichtige Prozesse
  - Selbstregulierungsmechanismen
- menschliche Schwächen
  - sich der Schwächen bewusst sein/werden
  - **Komplexität reduzieren**

Testen?

Testen ist Symptombekämpfung

- notwendig,
- aber unvollständig
- zu spät

Testen?

Test First!

- Ein Testfall pro Feature?
- Verbessert das Design
  → Behaviour-Driven Design (BDD)

Metriken?

Klassische Metriken:

- Coupling
- Vererbungstiefe
- Methoden pro Klasse
- zyklomatische Komplexität

↔ Lines Of Code?!

Khaled El Emam, Saida Benlarbi, Nishith Goel, and Shesh N. Rai: "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". IEEE Transasctions on Software Engineering, 27(7), July 2001

Metriken!

Metriken richtig interpretieren

- absolute Zahlen sind irreführend
- Entwicklungen beobachten
- Einzelfallbetrachtungen

Fahrplan

- Wie entstehen Fehler?
- Gegenmaßnahmen
- **Komplexität: Arten, Abgrenzung**
- Komplexität reduzieren
- Fazit

Komplexität – Definition

Aus der Systemtheorie:

„Die Komplexität eines Systems steigt mit

- der **Anzahl an Elementen**,
- der **Anzahl an Verknüpfungen** zwischen diesen Elementen
- sowie der **Funktionalität dieser Verknüpfungen**

(zum Beispiel **Nicht-Linearität**).“

Komplexität – Arten

- **Ontologische** vs **epistemologische** Komplexität
- **Komplexität** vs **Kompliziertheit**
- **inhärente** vs **hausgemachte** Komplexität

Kompliziertheit – Merkmale

- **viele** Elemente
- **viele** Verknüpfungen zwischen den Elementen
- für Verständnis **Expertenwissen** erforderlich
- aber: Verknüpfungen sind **statisch** und **linear**

Kompliziertheit – Maßnahmen

- Systemanalyse
- Divide and Conquer
- Zurückführen auf einfache Systeme
- Klassische Projektplanung

Komplexität – Merkmale

- **viele** Elemente
- **viele** Verknüpfungen
- Verknüpfungen **dynamisch** und **nicht-linear**
- Verhalten **prinzipiell nicht vorhersagbar**

Komplexität – Maßnahmen

- Den Blick aufs Ganze wahren
- Prototypisierung und Experimente
- Iterationen und Rückblicke
- Ständiges Dazulernen

→ **Agile Methoden**

Fahrplan

- Wie entstehen Fehler?
- Gegenmaßnahmen
- Komplexität: Arten, Abgrenzung
- **Komplexität reduzieren**
- Fazit

Kognitive Last reduzieren

menschliches RAM: **7 ± 2** „Dinge“

- Gedanken aufschreiben/loswerden
- Arbeitspunktlisten nutzen
- „Getting Things Done“
- Ablenkungen und Kontextwechsel vermeiden

→ **und: Code leicht lesbar gestalten!**

null

```
java.lang.NullPointerException

at org.libre.impress.Slide.showBullet(Slide.java:83)
at org.libre.impress.Slide.textArea(Slide.java:254)
at org.libre.impress.Slide.show(Slide.java:1035)
at org.libre.impress.Presentation.click(Presentation:32)
at org.libre.impress.Presentation.show(Presentation:123)
```

„I call it my billion-dollar mistake. It was the invention of the null reference in 1965.“
– Tony Hoare

Sonderfälle vermeiden

```
if (person != null)
    if (person.address != null)
        if (person.address.email != null)
            emailTo(person.address.email)

for (p ← person; a ← p.address; e ← a.email)
    emailTo(e)
```

Options verwenden!

→ Absicherung zur Compilezeit

Redundanz vermeiden (1)

```
@Test def showingNextSlideShouldWork = {
    val p = new Presentation(„slides.odp“)
    p gotoSlide 3
    assertThat(p.next.slideNumber) isEqualTo 4
    p.close
}
@Test def showingPreviousSlideShouldWork = {
    val p = new Presentation(„slides.odp“)
    p gotoSlide 6
    assertThat(p.prev.slideNumber) isEqualTo 5
    p.close
}
```

Redundanz vermeiden (2)

- Don't repeat yourself (DRY)
- Don't repeat yourself (DRY)
- also:
  - Unterfunktionen schreiben
  - Template Method Pattern

  → Higher Order Functions

Redundanz vermeiden (3)

```
// for logging purposes
// retrieve logger of class Bla
// change class name when copypasting!
Logger logger = Logger.getLogger(Bla.class);
```

```
/**
 * Get name of person.
 * @param person The person to get the name of
 * @return name The name to get
 */
 String getName(Person p) { return p.name; }
```

→ Kommentare sinnvoll einsetzen!

Redundanz vermeiden (4)

```
class Person {
    final String firstName;
    final String lastName;

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }
}
```

… was fehlt noch alles?

- equals() und hashCode()
- toString()

Redundanz vermeiden (5)

- Es geht auch anders...

```
case class Person(firstName: String,
                  lastName: String)
```

- **Moderne Programmiersprachen** verwenden!
- Notlösung für Java: Project Lombok

PLOP = Flop (1)

Hat jemand die Setter vermisst?

PLOP = Flop (2)

- PLOP: Place-Oriented Programming
  → Rich Hickey: The Value of Values

- Information besteht aus **unveränderlichen** Fakten
- Information hat eine **Zeitachse**

- Mutability ist ein Notbehelf
  - schlechte Angewohnheit, nicht Selbstzweck
  - heutzutage (meist) nicht mehr notwendig

PLOP = Flop (3)

Mutability führt zu:

- **Seiteneffekten**: Wer darf ein Objekt ändern?
- Locks et al: **Concurrency** sehr problematisch

Immutability führt zu:

- erhöhtem Speicherverbrauch
- Sorgenfreiheit
- niedrigerem Blutdruck

Bruchrechnung

```
interface Rational {
  void negate();
  void add(Rational other);
  void subtract(Rational other);
  (...)
}
```

```
interface Rational {
  Rational negate();
  Rational add(Rational other);
  Rational subtract(Rational other);
  (...)
}
```

Zustand vermeiden (1)

Objektzustand ist

- „versteckt“
- bestimmt aber das Verhalten

- Wie verstehen?
- Wie testen?
- Wie reproduzieren?

Oft unnötig!

Zustand vermeiden (2)

Wie entsteht Zustand?

- als Folge von **Ereignissen** (I/O)
- → nur für I/O notwendig

- → **Event Sourcing**
  - Nachvollziehbarkeit
  - Skalierbarkeit

Zustand vermeiden (3)

Zustand nach außen ziehen

- Geschäftslogik zustandsfrei halten
- ↔ Performance-Optimierung

- Zustand im Kleinen
  - Kapseln und Verstecken
  - Gut testen!
  - Von außen betrachtet: kein Zustand

Deklarativ programmieren

- „Reihenfolge“ führt zu Komplexität
  → *Moseley, Marks: Out of the Tar Pit*

- Reihenfolge oft irrelevant
  - Imperativität erzwingt Reihenfolge
  - Deklarativ programmieren!

→ deklarative DSLs einbetten

Gedanken direkt umsetzen (1)

„Summiere alle Zahlen bis *n*, die durch 3 oder 5 teilbar sind.“ (Project Euler, Aufgabe 1)

Steht da was von einer *for*-Schleife?

Da steht:

```
(1 to n) filter (x => x % 3 == 0 || x % 5 == 0) sum
```

Gedanken direkt umsetzen (2)

→ „Higher-Order Functions“

Wir denken in Higher-Order Functions,
nicht in Objektzuständen und Vererbungshierarchien.

Bsp: Concurrency

- `wait()`, `notify()`, `synchronized` – Locks und Monitore
- `java.util.concurrency` – Executors, Futures, Fork/Join
- akka: Aktoren, Agenten, Dataflow

Komplexität reduzieren – Fazit

- Kognitive Last reduzieren
- Sonderfälle vermeiden
- DRY – Redundanz vermeiden
- Immutability statt PLOP
- Zustand vermeiden
- Deklarativ programmieren
- High-Level nutzen

→ **Funktional programmieren!**

Fazit

- Wie entstehen Fehler?
  → durch Komplexität

- Gegenmaßnahmen
  → notwendige Komplexität adressieren
  → unnötige Komplexität reduzieren

Vielen Dank!

Joachim Hofer

imbus AG

@johofer, http://jmhofer.johoop.de

imbus AG

Spezialisierter Lösungsanbieter für Software-Qualitätssicherung und Software-Test

Innovativ seit 1992

Erfahrung und Know-how aus über 4.000 erfolgreichen Projekten

180 Mitarbeiter an vier Standorten in Deutschland

Beratung, Test-Services, Training, Tools, Datenqualität

Für den gesamten Software-Lebenszyklus

www.imbus.de