

DevSession: Altanwendungen wieder testbar machen

25.11.2019

Daniel Krämer, Maik Wolf

Daniel Krämer (M.Sc. Informatik)

- Software Engineer
- Schwerpunkte
 - Software-Architektur
 - Web Engineering
 - Integration und Migration
 - Clean Code
 - Testautomatisierung
 - Trainings, Vorträge, Artikel
- Java, Spring, Wicket, Microservices,...



Maik Wolf

- Software Engineer
- Schwerpunkte
 - Java Applikationen
 - Web Engineering
 - Continuous Delivery
- Java, JavaScript, Vaadin, Microservices, ...



anderScore GmbH

- Entwicklung von Individualsoftware
 - Consulting
 - Festpreis
- Kompletter Life Cycle
 - PM und RE
 - Architektur
 - Implementierung und Test
 - Wartung
- Trainings, Vorträge, Artikel
- Goldschmiede@anderScore

1. Leben mit dem Vermächtnis

6

2. Refactoring als Ausweg

10

3. Entwicklungsprozess

14

4. Techniken

15

5. Erfahrungsaustausch

19

6. Übung

21

7. Zusammenfassung

22

1. Leben mit dem Vermächtnis

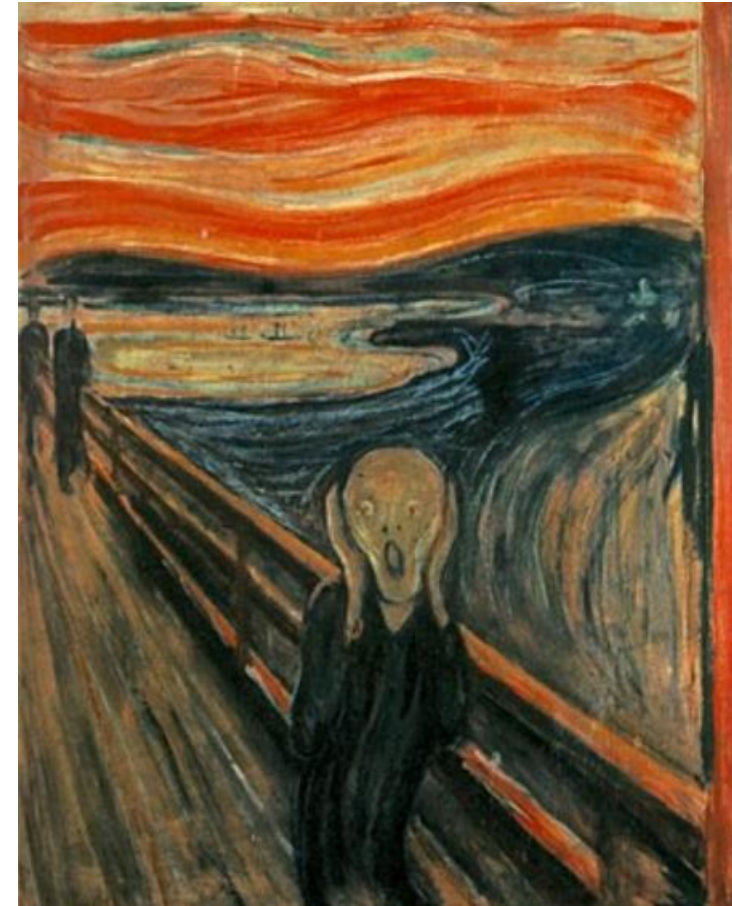
- *‘Kostbarkeiten’* geerbt von *‘Vorfahren’*
- Code *historisch gewachsen*
- Entwickler nicht mehr verfügbar
- Wartung erforderlich
- Wechselnde Verantwortlichkeiten
- Nicht einfach ersetzbar
- Oft: Probleme mit der Qualität...



- Vergangenheit: **Code & Fix**
 - Source Code
 - Schlecht designed
 - Schwer verständlich
 - Schwach dokumentiert und getestet
 - Nach Bedarf modifiziert
 - Probleme in Produktion
 - Wiederkehrend
 - Aufwendig zu reproduzieren
 - Schwierig zu analysieren
 - Dringlich und teuer in der Behebung



- Gegenwart: **‘Fear Driven Development’**
 - Anwendung faktisch unwartbar
 - Bugs zu fixen
 - Neue Features zu implementieren
 - Schmetterlingseffekte
 - Steigende Ansprüche
 - “... aber machen Sie bloß nichts kaputt!”
 - Schmerzen im Alltag eines jeden Entwicklers



- Zukunft: ???



- Zukunft: **kontinuierliches Refactoring !**
 - **Verständlichkeit erhöhen**
 - Aussagekräftige Bezeichner
 - Logische Strukturierung
 - Pragmatische Dokumentation
 - **Qualität des Codes verbessern**
 - Toter Code
 - Code Smells
 - Patterns und Konventionen
 - Error Handling

2. Refactoring als Ausweg

- Zukunft: **kontinuierliches Refactoring !**
 - **Testbarkeit wiederherstellen**
 - Code isolieren
 - Schnittstellen überarbeiten
 - Mocks einführen
 - Alles testen, was angefasst wird
 - Erst Unit Tests, dann Integration Tests
 - Ziel: stetiges Wachstum der Testbasis
 - **Wartbarkeit wiederherstellen**
 - Implementierung verstehen und testen
 - Neue Features mit Freude umsetzen
 - Erwartetes Verhalten sicherstellen



© Raimond Spekking / Wikimedia Commons, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=2693604>

- *“Unsere Kunden zahlen nicht für Refactoring!”*
 - ... aber sie sind offensichtlich bereit, deutlich mehr für Wartung und neue Features auszugeben!
- *“Das Projekt dauert sowieso nur noch ein paar Monate!”*
 - ... aber eine möglicherweise langjährige Wartungsphase schließt sich an!
- *“Wenden Sie Test Driven Development an!”*
 - ... aber die Anwendung ist nicht mehr testbar!
- *“Schreiben Sie Black Box Tests auf höherer Abstraktionsebene!”*
 - ... aber solche (Integration) Tests können sehr komplex werden...
 - ... und es ist schwer, an alle möglichen Konstellationen zu denken!

- Voraussetzungen
 - Collective Code Ownership
 - Ausschließlich **getesteter** und **eingesehener** Code
 - Kein Privatbesitz
 - Keine Wissensinseln
 - Keine Sklaventreiber
 - Unterstützung durch Tools
 - IDE, Refactoring Tools
 - Keine Veränderung der Geschäftslogik
 - Minimal möglicher Aufwand

Planung

1. Zusätzlichen Aufwand für Tests und Refactoring berücksichtigen

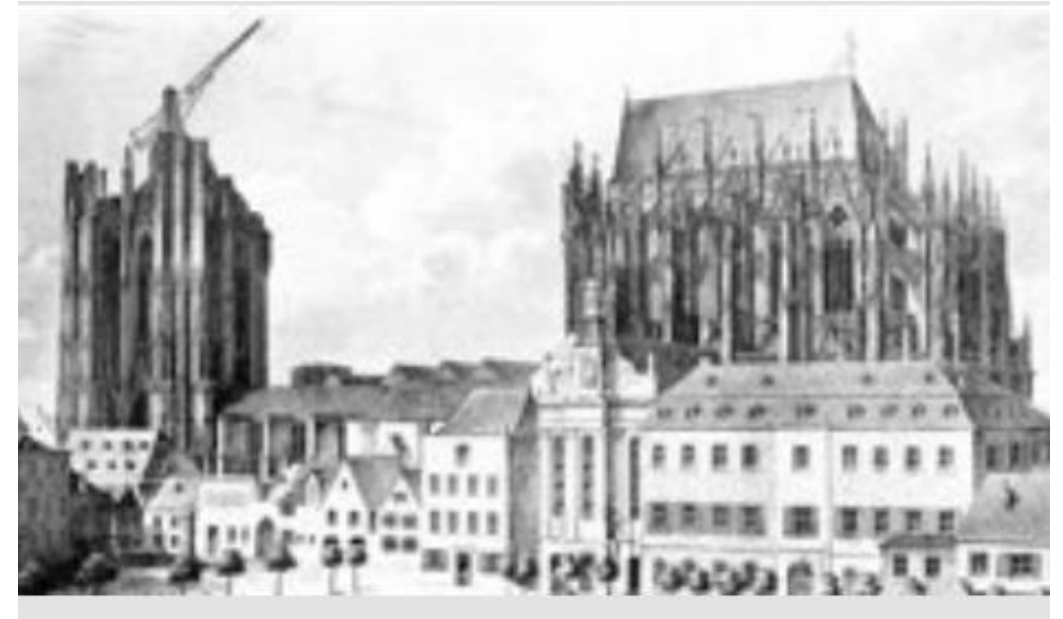
Vorbereitung

2. Nur strukturelle Refactorings durchführen
3. Überschaubare Regressionstests schreiben
4. Logische Refactorings durchführen
5. Tests verifizieren
6. Code Review durchführen

Entwicklung und Test

7. Neue Features einbauen
8. Tests verifizieren und erweitern

- Code extrahieren
 - Besseres Verständnis
 - Ermöglicht Testen in Isolation
 - Ermöglicht (partiell) Mocken
- Funktionaleren Programmierstil wählen
 - Klar definierte Schnittstellen
 - Vermeiden unerwarteter Seiteneffekte
 - Ermöglicht Black Box Tests
- Externe Abhängigkeiten reduzieren
 - Möglichst viele Daten als Eingabeparameter
 - Ermöglicht Mocken von Eingabedaten
 - Ermöglicht unterschiedliche Testszenarien



- Sichtbarkeit überdenken
 - Methoden sollten in sich geschlossen sein
 - Kontextabhängige Interfaces
 - Sichtbare Methoden leichter zu testen
- Integration Operation Segregation Principle anwenden
 - Operation: normale UnitTest
 - Integration: UnitTests mit Verifikation (Mocks)
- Schichten einführen
 - Spaghetti Code entwirren
 - Single Level of Abstraction
 - Unabhängiges Testen
 - Mocken auf verschiedenen Ebenen

Beispiel: vorher

```
private int getMyElementsForTheCalculations (int key2, String key1, List<Customer> list, int mode) {  
  
    // 1. Build SQL  
    // ...  
  
    // 2. Execute database call  
    // ...  
  
    // 3. Validate input parameters  
    // ...  
  
    if (error == 78) {  
        return 4711;  
    } else if (error != 321 && mode != 445 && Date.isToday("MO")) {  
        return -7;  
    }  
    return 1337;  
}
```



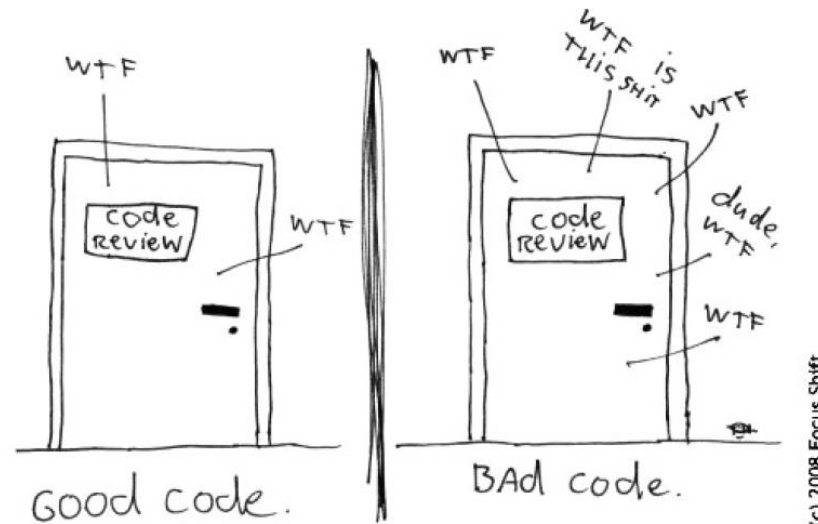
Beispiel: nachher

```
public List<Customer> getCurrentCustomers (String name, int zipCode) {  
  
    // 1. Validate input parameters  
    validateNameAndZipCode(name, zipCode);  
  
    // 2. Build SQL  
    String sql = createCustomerSelectSql(name, zipCode);  
  
    // 3. Execute database call  
    List<Customer> customers = database.query(sql);  
  
    return customers;  
}
```

5. Erfahrungsaustausch

- Entwickelt ihr mitunter auch Fear Driven?
- Was war euer letzter WTF-Moment?
- Welche Erfahrungen habt ihr mit Refactorings gemacht?

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



- Ausgangslage
 - “Altanwendung” auf Basis von Java, Spring, Wicket und Maven
 - Fachlichkeit: CRUD von Schedulern zum Abruf eines Ticket-Systems
 - Erweiterung des CSV Exports angefordert
- Probleme
 - CSV Export ist ein Big Ball of Mud...
 - Code weder verständlich, noch testbar
 - Entwickler nicht mehr im Haus

- Aufgaben
 - Exploration
 - Code Qualität verbessern
 - Testbarkeit herstellen
 - Reflektion
- Material
 - Folien, Aufgaben, Code
 - <https://github.com/anderscore-gmbh/JVM-Con-2019>

- **Problem:** Wartung von Software als 'Fear Driven Development'
- **Lösung:** kontinuierliches Refactoring auf *Designebene*
- **Ziel:** Altanwendungen wieder testbar (und wartbar) machen
- **Voraussetzungen:** Collective Code Ownership, Tools
- **Umsetzung:** Einbettung in Entwicklungsprozess
- **Handwerkszeug:** Refactoring-Techniken
- **Grenzen:** architekturelle, technologische oder organisatorische Probleme



- Refactoring Legacy Code: <http://refactoring-legacy-code.net>
- Refactoring Guru: <https://refactoring.guru>
- Refactoring Catalog: <https://refactoring.com/catalog>
- Mockito: <http://site.mockito.org>

- Fowler, et. al.: Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional
- Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall

- Bilder: Bing Bildersuche (verschiedene Sites; Creative Commons)