Goldschmiede anderScore

Legacy Software wieder testbar machen

02.02.2018, Daniel Krämer

© 2018 anderScore GmbH

Vorstellung

anderScore trust in competence

Daniel Krämer (M.Sc. C.S.)

- Software-Entwickler
- Schwerpunkte
  - Java
  - Web Engineering
  - Software-Architektur
  - Migration und Integration
  - Clean Code
  - Testautomatisierung
- Trainings (Wicket, Spring, jQuery)
- Vorträge, Artikel

Inhalt

1. Leben mit dem Vermächtnis

- *‘Kostbarkeiten’* geerbt von ‘*Vorfahren’*

- Code *historisch gewachsen*
- Entwickler nicht mehr verfügbar
- Wartung erforderlich
- Wechselnde Verantwortlichkeiten
- Nicht einfach ersetzbar

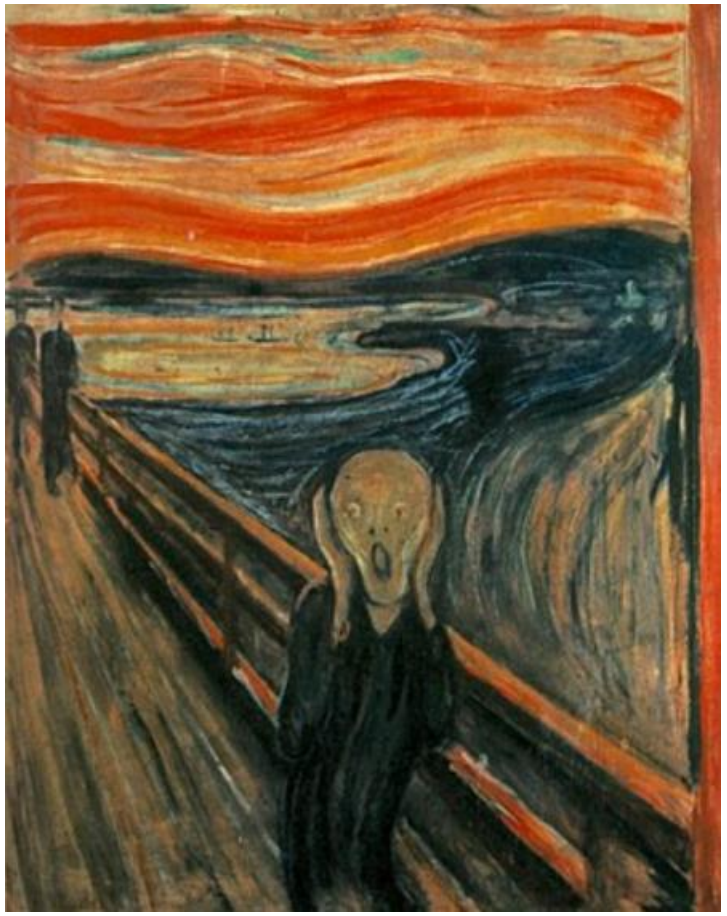
- Oft: Probleme mit der Qualität…

1. Leben mit dem Vermächtnis

- 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

1. Leben mit dem Vermächtnis

- 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



1. Leben mit dem Vermächtnis

- Zukunft: **???**

2. Refactoring als Ausweg

- 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

2. Refactoring als Ausweg

- “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!

2. Refactoring als Ausweg

anderScore trust in competence

- 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

3. Entwicklungsprozess

Planung

1. Zusätzlichen Aufwand für Tests und Refactoring berücksichtigen

Vorbereitung

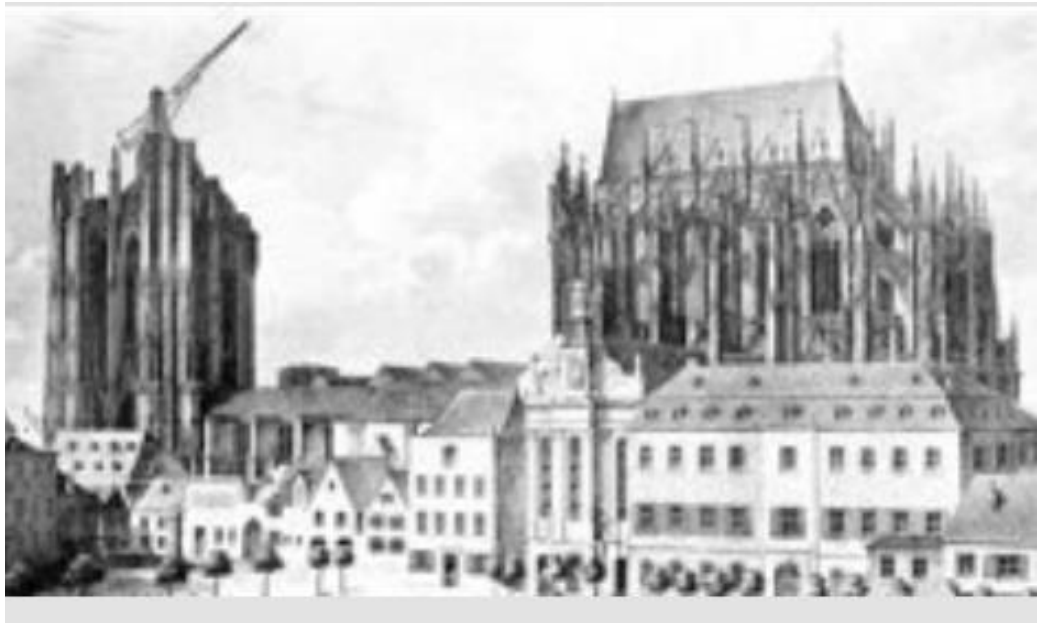
2. Nur strukturelle Refactorings durchführen
3. Überschaubare Regressionstests schreiben
4. Logische Refactorings durchführen
5. Tests verifizieren
6. Code Review durchführen

Entwicklung und Test

7. Neue Features einbauen
8. Tests verifizieren und erweitern

4. Techniken

- Code extrahieren
  - Besseres Verständnis
  - ➢ Ermöglicht Testen in Isolation
  - ➢ Ermöglicht (partielles) 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



4. Techniken

- 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

anderScore trust in competence

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;
}
```

anderScore trust in competence

Beispiel: nachher

```
public List<Customer> getActiveCustomers (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. Zusammenfassung

- **Problem:** Wartung von Software als ‘Fear Driven Development’
- **Lösung:** kontinuierliches Refactoring auf *Designebene*
- **Ziel:** Legacy Software wieder testbar (und wartbar) machen

- **Voraussetzungen:** Collective Code Ownership, Tools
- **Umsetzung:** Einbettung in Entwicklungsprozess
- **Handwerkszeug:** Refactoring-Techniken

- **Grenzen:** architekturelle, technologische oder organisatorische Probleme

Literatur

- 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

Quellen

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