



Coding Dojo – Der Nutzen von TDD

27. Juli 2016

Ursachen

- ▶ Ursprünglich geplantes Vorgehen
- ▶ Probleme
- ▶ Kurzfristige und repräsentative Lösung

Zeitplan

09⁰⁰-10⁰⁰ Uhr: Begrüßung

10⁰⁰-12⁰⁰ Uhr: Beginn der Implementierung

12⁰⁰ -12³⁰ Uhr: Review

12³⁰-13⁰⁰ Uhr: Mittagessen

13⁰⁰-13³⁰ Uhr: Retrospektive

Ablauf

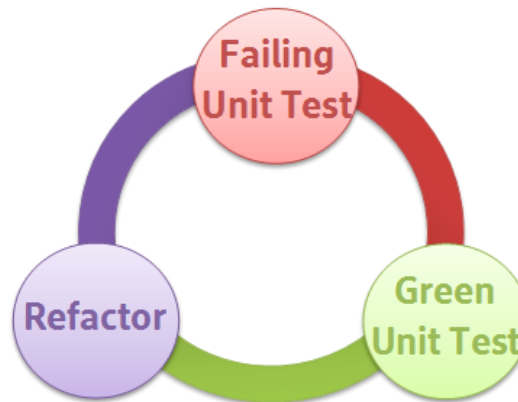
- ▶ Klärung der wichtigsten Begriffe und Rahmenbedingungen
- ▶ Vorstellung der Kata
- ▶ Einteilung in zwei 2 Gruppen
 - ▶ Eine Gruppe verwendet TDD
 - ▶ Eine Gruppe verwendet gar keine Tests (Main-Methode erlaubt)
- ▶ Bescheid geben, sobald Aufgabe gelöst ist

Was ist ein Coding Dojo?

- ▶ Trainingsort für asiatische Kampfsportarten
 - ▶ Dojo bedeutet Weg und Ort
 - ▶ Somit ist ein Dojo ein Ort an dem der Weg geübt wird
- => Coding Dojo: Ort an dem der Weg des Programmierens geübt wird
-
- ▶ Kata: Übungsmethodik
 - ▶ Unabhängige Programmieraufgabe
 - ▶ Erlernen einer bestimmten Technik
 - ▶ Babysteps

Was ist Test-Driven Development?

- ▶ Ablauf:
 - ▶ 1. Test schreiben – **Red**
 - ▶ 2. Test durch Implementierung der gewünschten Funktionalität erfolgreich durchlaufen lassen – **Green**
 - ▶ 3. Code gegebenenfalls überarbeiten und verbessern – **Refactor**



Was ist Test-Driven Development?

- ▶ Funktionsgetriebener Ansatz
- ▶ Höhere Testabdeckung
- ▶ Unit-Tests:
 - ▶ Minimierung des manuellen Testaufwands
 - ▶ Bessere Wartbarkeit
 - ▶ Weniger Fehler
 - ▶ Absicherung

Pair Programming – Driver Navigator

► Rolle des Driver:

- Bedient Tastatur und Maus
- Schreibt den Code nach Anweisung des Navigators
- Konzentriert sich auf die aktuelle Aufgabe
- Frägt nach, wenn er etwas nicht versteht
- Vertraut dem Navigator, gibt aber alternative Vorschläge

► Rolle des Navigator:

- Gibt dem Driver klare Anweisungen
- Hat das gesamte Problem im Blick
- Begründet seine Lösungsansätze
- Prüft den Code/Liest mit
- Bleibt aufmerksam, beteiligt sich AKTIV!

⇒ Rollenwechsel nach jedem grünen Test-Durchlauf

SOLID-Prinzipien

- ▶ **Single Responsibility Principle:**
 - ▶ Jede Klasse sollte nur eine Verantwortlichkeit haben
 - ▶ Eine Klasse sollte immer nur einen einzigen Grund haben, sich zu ändern
- ▶ **Open/Closed Principle:**
 - ▶ Klassen sollten für Erweiterungen offen sein, aber geschlossen bezüglich Veränderungen
 - ▶ Erweiterung z.B. durch Vererbung

SOLID-Prinzipien

- ▶ **Liskov's Substitution Principle:**
 - ▶ Objekte sollten durch Instanzen ihrer Subtypen ersetzbar sein
 - ▶ Subtyp darf Funktionalität nur erweitern
- ▶ **Interface Segregation Principle:**
 - ▶ Spezifische Interfaces statt einem Allround-Interface
- ▶ **Dependency Inversion Principle:**
 - ▶ Klassen höherer Ebenen sollten nicht von Klassen niedrigerer Ebenen abhängig sein, sondern von Interfaces