LU07a - Was ist Refactoring

Refactoring heisst: die *innere Struktur* von bestehendem Code verbessern, ohne sein *nach aussen sichtbares Verhalten* zu verändern. Gleiche Eingabe, gleiche Ausgabe, gleiche Seiteneffekte - nur der Weg dorthin ist danach lesbarer, kürzer oder besser zerlegt.

Die Faustregel: Nach einem Refactoring muss jeder Test, der vorher grün war, immer noch grün sein. Wenn sich das Verhalten ändert, war es kein Refactoring - dann haben Sie entweder einen Bug behoben oder einen neuen eingebaut.

Abgrenzung: was Refactoring nicht ist

Der Begriff wird im Alltag für alles verwendet, was man an altem Code tut. Für Band D müssen Sie die Fälle auseinanderhalten:

Tätigkeit	Verhalten ändert sich	Struktur ändert sich	Beispiel
Refactoring	nein	ja	Eine 60-Zeilen-Funktion in vier benannte Funktionen zerlegen
Bugfix	ja (das falsche Verhalten verschwindet)	evtl.	< durch ⇐ ersetzen, weil die Grenze falsch war
Neues Feature	ja (es kommt etwas dazu)	ja	Eine Exportfunktion ergänzen
Optimierung	nein (Ergebnis gleich, Laufzeit anders)	ja	Eine Liste durch ein `set` ersetzen
Rewrite	oft ja	vollständig	Das Modul wegwerfen und neu schreiben

Refactoring und Optimierung sehen ähnlich aus, verfolgen aber verschiedene Ziele: Refactoring optimiert für den **Menschen**, der den Code liest, Optimierung für die **Maschine**, die ihn ausführt. Beide gehören zum Kompetenzband D, deshalb behandelt diese Learning Unit beide - zuerst das Lesbare (LU07b bis LU07e), dann das Schnelle (LU07f und LU07g).

Nie beides gleichzeitig. Wer im selben Schritt umbaut *und* ein Feature ergänzt, kann bei einem Fehler nicht mehr sagen, welche der beiden Änderungen ihn verursacht hat. Erst refactoren, testen, committen - dann das Feature.

Warum überhaupt?

Code wird viel häufiger gelesen als geschrieben. Jede Minute, die eine unklare Funktion beim nächsten Lesen kostet, zahlen Sie mehrfach - und in einer Projektarbeit oft schon nach zwei Wochen an sich selbst.

- **Verständlichkeit:** Der Code erklärt sich durch Namen und Struktur, statt durch Kommentare erklärt werden zu müssen.

- **Änderbarkeit:** Eine Anpassung betrifft eine Stelle statt fünf Kopien.
- **Testbarkeit:** Kleine, pure Funktionen lassen sich einzeln prüfen, eine 60-Zeilen-Funktion mit globalem Zustand nicht.
- **Fehlersuche:** Wo die Zerlegung stimmt, ist der Fehler in wenigen Zeilen eingekreist.

Der Refactoring-Zyklus

Refactoring ist keine grosse Aufräumaktion am Freitagnachmittag, sondern eine Folge sehr kleiner Schritte:

1. **Sicherheitsnetz prüfen.** Gibt es Tests, die das aktuelle Verhalten festhalten? Wenn nein: zuerst welche schreiben (siehe LU07e).
2. **Einen Smell auswählen.** Genau einen, nicht fünf (siehe LU07b).
3. **Eine Technik anwenden.** Möglichst mit dem Refactoring-Werkzeug der IDE statt von Hand.
4. **Tests laufen lassen.** Rot? Sofort zurück zum letzten funktionierenden Stand.
5. **Committen.** Kleine Commits sind später Ihr Nachweis, dass das Verhalten gleich geblieben ist.
6. Zurück zu Schritt 2.

Faustregel für die Schrittgrösse: Wenn Sie nach einer Änderung länger als ein paar Minuten brauchen, um den Code wieder lauffähig zu bekommen, war der Schritt zu gross.

Wann refactoren Sie?

- **Vorbereitend**, bevor Sie ein Feature einbauen: Erst den Code so umstellen, dass die Erweiterung leicht wird - dann die Erweiterung einbauen.
- **Aufräumend**, direkt nachdem etwas funktioniert: Der erste lauffähige Wurf darf hässlich sein, muss es aber nicht bleiben.
- **Nebenbei** (Pfadfinderregel): Jede Datei, die Sie anfassen, verlassen Sie etwas sauberer, als Sie sie vorgefunden haben.

Nicht refactoren sollten Sie am Tag vor einer Abgabe ohne Tests, an Code, den niemand mehr liest, und an Stellen, die Sie nicht verstehen - dort steht zuerst Lesen und Testen an.

Einordnung ins Kompetenzband D

Diese Learning Unit deckt das Kompetenzband **D - Refactoring und bestehenden Code optimieren** ab:

Feld	Anspruch	Seiten
D1B	Techniken und Massnahmen **aufzählen** und deren Zweck **erklären**	LU07a, LU07b, LU07c
D1I	Techniken **anwenden**, vorgegebene Massnahmen **umsetzen**	LU07d, LU07e, LU07f
D1A	Auswirkungen **einschätzen**, Nebeneffekte **vermeiden**, Datenstruktur **auswählen**	LU07e, LU07g

Im Portfolio (LB02) verlangt jedes dieser drei Felder ein Pflicht-Artefakt aus Ihrem eigenen Flask-Projekt. Sammeln Sie ab jetzt Vorher/Nachher-Stände: ein sauberer Commit ist später der Nachweis,

den Sie sonst nachträglich konstruieren müssen.

Quellen

- Martin Fowler: *Refactoring. Improving the Design of Existing Code*, 2. Auflage
- Online-Katalog der Refactoring-Techniken

M323-LU07, M323-D1B

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m323/learningunits/lu07/grundlagen

Last update: **2026/09/09 11:06**