**Kapitel 8: Design und Refactoring – Genauer erklärt**

**1. Warum ist gutes Design wichtig?**

* In kleinen Projekten kann man schnell und unkompliziert Code schreiben – Hauptsache, es funktioniert.
* In größeren Projekten (wie echten Data-Science-Anwendungen) wird der Code jedoch schnell **unübersichtlich** und **schwer wartbar**, wenn er nicht gut strukturiert ist.
* Gutes Design bedeutet: Der Code ist **klar gegliedert**, **leicht erweiterbar** und **verständlich** – auch für andere (oder für dich selbst in ein paar Monaten).

➡️ **Gutes Design spart langfristig Zeit und Nerven**

**2. Von Notebooks zu Skripten**

* Data Scientists arbeiten oft mit **Jupyter-Notebooks**, weil sie ideal für **explorative Analysen** sind: Schnell ausprobieren, direkt Ergebnisse sehen.
* Aber: Notebooks werden schnell chaotisch und sind **schlecht geeignet** für robuste, wiederverwendbare Software.

**Der Übergang, den Nelson beschreibt:**

* **Schritt 1**: Den funktionierenden Notebook-Code aufteilen in **klare Funktionen** (z.B. load\_data(), train\_model(), evaluate\_model()).
* **Schritt 2**: Diese Funktionen in **Skripte oder Python-Module** auslagern (.py-Dateien).
* **Schritt 3**: Die Hauptlogik in ein **Hauptskript** oder eine **Kommandozeilen-Anwendung** packen.
* **Schritt 4**: Tests schreiben, damit spätere Änderungen sicher überprüft werden können.

➡️ Ziel: **Produktionsreife Anwendungen** statt chaotische Experimente.

**3. Wichtige Prinzipien für gutes Design**

Hier bringt Nelson klassische Software-Engineering-Prinzipien in die Data-Science-Welt:

| **Prinzip** | **Bedeutung** |
| --- | --- |
| **Modularität** | Code wird in kleine, unabhängige Einheiten (Funktionen, Klassen) aufgeteilt. Jede Einheit macht **eine** Sache gut. |
| **DRY (Don't Repeat Yourself)** | Vermeide es, denselben Code an mehreren Stellen zu kopieren. |
| **Lesbarkeit** | Code sollte so geschrieben sein, dass jemand anders ihn **leicht verstehen** kann (gute Namen, klare Struktur). |
| **Effizienz** | Besonders bei großen Datenmengen: Schreibe Code, der ressourcenschonend ist. |
| **Robustheit** | Dein Code sollte mit unerwarteten oder fehlerhaften Eingaben umgehen können, ohne abzustürzen. |

**4. Was ist Refactoring genau?**

**Definition:**  
➔ *Refactoring ist die Umstrukturierung von Code, um dessen Qualität zu verbessern –* ***ohne*** *die Funktionalität zu verändern.*

**Warum Refactoring?**

* Weil erster Entwurfs-Code oft schnell und unsauber geschrieben wird.
* Weil sich Anforderungen ändern (z.B. neue Features) und der alte Code angepasst werden muss.

**Typische Refactoring-Aktivitäten:**

* Aufteilen von langen Funktionen in kleinere, besser verständliche Funktionen.
* Umbenennen von Variablen und Funktionen für bessere Verständlichkeit.
* Entfernen von doppeltem Code.
* Ersetzen von Copy-Paste-Logik durch generische Funktionen.
* Strukturieren von Code in Module und Pakete.

**Wichtig:**  
Vor dem Refactoring sollte der Code **vollständig getestet** sein, damit du nach dem Umbau sicherstellen kannst, dass alles noch funktioniert.

**5. Code Smells – Warnzeichen für schlechtes Design**

Catherine Nelson nennt Beispiele für typische „Code Smells“ (Probleme, die auf schlechtes Design hinweisen):

* **Duplizierter Code** (mehrfach dieselbe Logik)
* **Zu lange Funktionen** (eine Funktion sollte klar umrissen sein)
* **Unklare Namen** (z.B. foo, data1 statt calculate\_mean\_temperature)
* **Komplexe Bedingungen** (zu verschachtelte if-else-Logik)
* **Verletzung des Single-Responsibility-Prinzips** (eine Funktion/Klasse macht zu viele verschiedene Dinge)

➡️ **Wenn du solche Anzeichen siehst, solltest du über ein Refactoring nachdenken.**

**6. Tipps für den Refactoring-Prozess**

* **Schrittweise vorgehen**: Nicht alles auf einmal umbauen, sondern in kleinen, nachvollziehbaren Schritten.
* **Zwischendurch testen**: Nach jedem Schritt prüfen, ob der Code noch korrekt funktioniert.
* **Mut zur Verbesserung**: Auch wenn der Code "funktioniert", lohnt sich manchmal das Aufräumen, damit er langfristig besser wird.