Kriterien für Programmqualität

Nicht *nur* Korrektheit

Lesbarkeit	Ist nachvollziehbar was das Programm tut?
Testbarkeit	Ist es einfach zu testen ob das Programm sich korrekt verhält?
Änderbarkeit	Ist es einfach das Programm anzupassen sobald die Anforderungen sich ändern?
Wiederverwertbarkeit	Sind die Teile des Programms strukturell gut geeignet wiederverwendet zu werden sobald Programm geändert/neues Programm entwickelt werden muss?

Hohe Qualität ⟶ langfristig weniger Fehler, leichtere Fehlersuche, schnellere Entwicklung

Begriff: Pattern/Anti-Pattern

Pattern	Beobachtete Muster in Code hoher Qualität
Anti-Pattern	(auch *Code smells*) Beobachtete Muster in Code niedriger Qualität

- Einzelne Anti-/Patterns oft Programmiersprachen- und Konzept-übergreifend (Bibliotheken/Klassen/Methoden/Funktionen/…)
- Keine komplett starren Muster; nur teilweise Umsetzung der Empfehlungen oft schon nützlich
- Oft nicht unabhängig voneinander
- Patterns verwenden, Anti-Patterns vermeiden

Pattern: Kurzer Code

- Kürzerer Code bei gleicher Funktionalität oft besser
- Ziel sind features, nicht lines of code
- Code der nicht existiert hat auch keine bugs
- Weniger Code ist automatisch übersichtlicher
- Konziser Code jedoch gelegentlich auch schlechter verständlich

Pattern: Resource Acquisition Is Initialisation (RAII)

- *Ressource* (*Betriebsmittel*) ist alles was nur begrenzt zur Verfügung steht (Arbeitsspeicher, CPU-Zeit, Dateien, Netzwerk, …)
- Ressourcen werden vom Betriebssystem verwaltet
  Programm akquiriert Ressourcen; müssen oft wieder freigegeben werden (Speicher, Dateien, …) sonst Probleme (Speicherleck, …)
- Freigabe wird jedoch oft vergessen
- Ansatz: akquiriere Ressource in Konstruktor, Freigabe in Destruktor ⟶ Verknüpft Ressource mit Lebensdauer eines Objekts sodass Freigabe nicht vergessen werden kann (Destruktoraufruf i.d.R. implizit)
- Leitmotiv von C++

Einschub: Smart Pointer

- Standardlibrary stellt in `<memory>` Klasse `unique_ptr<T>`
- Konstruktor nimmt Zeiger als Parameter (jedoch `explicit`, keine implizite Typkonvertierung)
- Destruktor ruft `delete` auf Zeiger auf
- Objekte von `unique_ptr<T>` *nicht kopierbar*

Array auf dem Heap

```
int main() {
  unique_ptr<int[]> v{new int[1000]};

  for (int i = 0; i < 1000; i++)
    v[i] = i;
}
```

```
…
==…== All heap blocks were freed --
↪  no leaks are possible
…
```

Pattern: Separation of Concerns

- Saubere Trennung zwischen Programmteilen
- Keine Abhängigkeiten von z.B. *interner Darstellung* anderer Datenstrukturen
- Je komplexer das Problem, desto wertvoller ist feine Aufteilung
- Faustregel für gute Aufteilung: kann ich die Aufgabe eines Programmteils in natürlicher Sprache *kurz* beschreiben?

Anti-Pattern: Feature Envy

- Ein Programmteil A implementiert Funktionalität die eigentlich in anderen Programmteil B gehört
- B sollte stattdessen ein geeignet abstraktes feature haben für diesen Anwendungszweck
- Verletzt separation of concerns

```
Matrix m;
double wert = 7.2;
int x = 3, y = 17;
m.liste[y * 5 + x] = wert;
```

```
Matrix m;
double wert = 7.2;
int x = 3, y = 17;
m.getEntry(x, y) = wert;
```

Pattern: Design by Contract

- Programm aufteilen in kleine Teile (Klassen, Methoden)
- Teile separat voneinander entwickeln *und testen*
- Teile bekommen *vor*/bei Implementierung klar definierte Aufgaben, Eigenschaften, Erwartungen und Verhalten (gemeinsam der *contract*)
  ⟶ Teile werden austauschbar
- Setze größere Programmteile (am Ende das ganze Programm) zusammen aus kleineren Teilen
- Idee: wenn alle kleinen Teile ihrer Spezifikation genügen, dann auch das ganze Programm

Design by Contract: Entwicklung von Teilen

- In objektorientierten Sprachen (C++, Java, …) sind Klassen angedacht für die Rolle als Teile
- Vor/Beim Implementieren überlegen und *dokumentieren*:
  - Welche Leistungen erbringen Methoden der Klasse/die Klasse insgesamt?
    I.d.R. in der Form von *Nachbedingungen* und Beschreibung der Effekte der Methode
  - Welche Annahmen machen Methoden über Zustand des Objekts und ihre Parameter?
    I.d.R. in der Form von *Vorbedingungen*
- Nur dokumentierte Leistungen verwenden und stets sicherstellen, dass dokumentierte Annahmen erfüllt sind
- *Klasseninvarianten* sind Vorbedingung und Nachbedingung für alle Methoden (für Konstruktoren nur Nachbedingung)

Einschub: `assert`

- Ideal: Typen der Parameter aller Methoden *garnicht in der Lage* invalide Werte auszudrücken
- Realistischer: `public` Methoden überprüfen ihre Parameter und werfen exceptions wenn invalide
- Parameter an `private` Methoden jedoch gut unter Kontrolle – diverse Überprüfungen scheinen verschwenderisch
- `<cassert>` enthält *Präprozessor-Makro* `assert`; für b Ausdruck vom Typ `bool`, verwende `assert((`b`))`
- b wird zur Laufzeit ausgewertet, bricht Ausführung ab falls `false`
- Übersetzung mit Compiler-Kommandozeilenparameter `-DNDEBUG` entfernt alle `asserts`

Beispiel: `assert`

assert.cpp

```cpp
#include <iostream>
#include <cassert>

using namespace std;

int main() {
  assert((1 == 2));
  cout << "output" << endl;
}
```

```
g++ assert.cpp; ./a.out
```

```
assert: _copy/assert.cpp:7: int
↪  main(): Assertion `(1 == 2)'
↪  failed.
```

```
g++ -DNDEBUG assert.cpp; ./a.out
```

```
output
```

Anti-Pattern: Code Duplikation

- Code wird an verschieden Stellen kopiert, evtl. jeweils leicht angepasst
- Unübersichtlich und unleserlich
- Änderungen werden leicht inkonsistent, werden oft nicht korrekt an alle anderen Stellen übertragen wo i.W. identischer Code vorkommt
- Relevante Unterschiede gehen unter in viel umliegendem kopiertem Code und werden schwer erkennbar
- Stattdessen Code-Struktur überdenken; Code in geteilte Funktionen auslagern

Pattern: Don't Repeat Yourself (DRY)

- Jedes feature und jedes Stück *Wissen* (Konstanten/Algorithmen/…) sollte an *einer* Stelle im Programm repräsentiert sein
- Sobald sich Anforderungen ändern muss nur *die eine Stelle* geändert werden
- Vermeidet Inkonsistenz innerhalb des Programms und damit Fehler

Anti-Pattern: Lange Funktionen/Methoden

- Zu lange Funktionen/Methoden sind oft Symptom anderer Probleme:
  - Zu viel Funktionalität ⟹ schlechte Wiederverwertbarkeit
  - Code-Duplikation
  - Umständliche Lösung ⟹ oft schlechtere Lesbarkeit, mehr Gelegenheiten für Fehler
- Kürzere Funktionen/Methoden i.d.R. einfacher zu verstehen
- Funktionen/Methoden vergleichsweise einfach separat zu testen ⟹ umfangreiche Funktionen/Methoden erschweren Fehlersuche

Anti-Pattern: Zu große Klassen

- Auch *Gott-Objekt*
- Probleme analog zu denen zu langer Funktion/Methode
- Programmfluss durch umfangreiche Klasse oft schwer nachzuvollziehen
- Zusätzlich oft Symptom schlechter Rollenverteilung/unsauberer Trennung von Programmteilen

Anti-Pattern: Viele Variablen

Oft Symptom für:

- Code-Duplikation; Zwischenwerte evtl. mehrfach berechnet mit anderen Namen?
- Berechnungsablauf unsauber definiert/strukturiert
- Zu große Funktion/Methode/Klasse
- Ergebnisse von Berechnungen werden nicht (mehr) verwendet

Pattern: Wenige Variablen, kleiner Gültigkeitsbereich

Möglichst wenige Variablen im aktuellen Gültigkeitsbereich:

- Einfachere Übersicht über alle relevanten Variablen
- Kürzere Einlesezeit
- Reduziert versehentliche Verwechslungen

Hierfür:

- Immer minimalen Gültigkeitsbereich wählen
  - Keine globalen Variablen verwenden
  - Schleifenvariablen in Schleife deklarieren
  - Zwischenergebnisse von Methoden nicht in Objekt-Attributen speichern
  - Variablen bei erster Verwendung deklarieren; nicht davor
- Siehe „Viele Variablen“, „Code Duplikation“

Pattern: Gute Namen

- Name der Variable sollte ihre *Funktion* beschreiben nicht Inhalt (`const int BITS = 8;` nicht `const int EIGHT = 8`)
- Mittlere Länge oft am Besten
  - Zu kurz (1-3 Zeichen) nicht aussagekräftig
  - Zu lang behindert Lesefluss
- Je kleiner der Gültigkeitsbereich, desto besser sind kurze Namen gerechtfertigt (z.B. Schleifenvariable `i`)

Anti-Pattern: Hungarian Notation/Name typing

```
set<Student> getSetOfAllStudents();
map<Seminar, set<Student>> studentSetPerSeminarMap;
```

Nach der nächsten Änderung:

```
list<Student> getSetOfAllStudents();
map<Seminar, list<Student>> studentSetPerSeminarMap;
```

Typannotationen bereits hinreichend; Kodierung zusätzlich im Namen redundant

Pattern: Gut passender Typ

- Bei Einführen einer Variable/Deklarieren einer Datenstruktur stets Typ wählen, der zu späterem Inhalt und Verwendung möglichst exakt passt
- Z.B. vorzeichenlose Zahlentypen wo möglich
- Lieber ABC (siehe Vererbung) statt konkreter davon abgeleiteter Implementierung
- *Make invalid states unrepresentable*

Anti-Pattern: Cargo Cult Programming

Dinge verwenden, die man nicht wirklich verstanden hat

- (Anti-)Patterns beachten ohne Gründe zu verstehen
- Code aus anderer Stelle kopieren und verwenden

Unverstandenen Code kann man auch nicht debuggen