Designprinzipien

1. Single Responsibility Principle (SRP)
   - Unix-Philosophie: *«Do only one thing and do it well.»*
   - Eine Klasse hat nur *eine* Zuständigkeit
   - Eine Klasse hat nur *einen* Grund sich zu ändern
   - Änderungen/Erweiterungen beschränken sich auf möglichst wenige Klassen
   - Viele kleine Klassen statt wenige grosse Klassen → verbesserte Testbarkeit
2. Don't Repeat Yourself (DRY)
   - Duplizierung von Code vermeiden
     - Einfache Duplizierung: kopierte Codefragmente
     - Subtile Duplizierung: wiederholte Bedingungskonstrukte, identische Algorithmen unterschiedlich implementiert
   - Lösung: Unterfunktionen, Objektorientierung, Design-Patterns
   - Vermeidung von Duplizierung verbessert/ermöglicht Wiederverwendung und erhöht dadurch die Produktivität
3. Favor Composition over Inheritance (FCoI)
   - Komposition ist flexibler und weniger wartungsintensiv
     - Wrapper-, Delegate- oder Decorator-Pattern (GoF)
     - Komposition ist ohne Einfluss auf das Interface austauschbar
   - Im Zweifelsfall Komposition der Vererbung vorziehen
   - Nicht von Klassen erben, die nicht explizit für Vererbung vorgesehen sind (selbst wenn diese *nicht* mit `final` deklariert sind
4. Design for Inheritance, or Prohibit it
   - Das Schreiben gut spezialisierbarer Basisklassen ist schwierig
   - Implementation muss für API dokumentiert werden → Verletzung der Datenkapselung
   - Aufruf von überschreibbaren Methoden ist gefährlich
   - Klassen sollten entweder zum Überschreiben entworfen werden oder als `final` deklariert werden bzw. private Konstruktoren haben
5. Hohe Kohäsion, lose Kopplung
   - Hohe Kohäsion: eine Klasse fasst nur Attribute und Methoden zusammen, die wirklich zusammengehören
     - Optimale Kohäsion ist erreicht, wenn eine weitere Teilung nicht mehr sinnvoll ist
   - Lose Kopplung: Wenige Abhängigkeiten zwischen Klassen
6. Datenkapselung vs. Information Hiding
   - Datenkapselung: expliziter und bewusster Umgang mit Zugriffsmodifikatoren
     - Attribute möglichst `private`, aber *nicht* alle Methoden `public`
     - private Attribute mit öffentlichen getter- und setter-Methoden reicht nicht
   - Information Hiding: strikte Trennung von Schnittstelle und Imple-

mentierung
    - Schnittstelle sollte keine Rückschlüsse auf die interne Implementierung ermöglichen
  - fördert hohe Kohäsion und lose Kopplung
7. Schnittstellen zur Enkopplung und Abstraktion
  - Schnittstellen als Mittel zur Abstraktion und Entkopplung
  - Schnittstellen als bessere Alternative zu abstrakten Klassen
  - *«Design by Interface»*: Fokus auf das *was* (Schnittstelle), dann auf das *wie* (Implementierung)
    - Fokus auf die Perspektive des Nutzers
  - Im Zweifelsfall lieber zu viele als zu wenige Schnittstellen
  - Erleichtert Einhaltung des Test-First-Principles
8. Test First Principle
  - Nicht nachträglich testen um Fehler zu finden
  - Stattdessen fortlaufend testen, um Gewissheit zu haben, dass es funktioniert
  - Automatisiertes Testen (mit dem JUnit-Framework) ist sehr effizient
  - Test First Programming/Test-Driven Development
    - Früh die Perspektive des Nutzers einer Klasse einnehmen
    - Intuitive Aufdeckung und Berücksichtigung von Spezialfällen
  - Kein Projekt ist zu klein, um nicht mit Unit-Tests getestet zu werden
9. viele kleine statt wenige grosse Einheiten
  - Einheiten: Klassen und Methoden
  - Kleine Einheiten sind besser verständlich, wiederverwendbar und testbar
  - Bei jeder Erweiterung überprüfen, ob nicht eine Aufteilung besser wäre
    - Kurzfristig höherer Arbeitsaufwand, langfristig geringerer Arbeitsaufwand
  - Refactoring als andauernder Prozess (permanentes Refactoring)
10. aussagestarke Namen und formatierter Quellcode
  - Klassen, Methoden usw. nicht leichtfertig benennen
  - Ein guter Name hilft
    - Sinn und Absicht zu verstehen
    - Dokumentationsaufwand zu reduzieren
    - Verletzungen der Kohäsion zu erkennen: `doThisAndThat()`
    - beim Verständnis des Quellcodes (bessere Wartbarkeit und Erweiterbarkeit)
  - Quellcode immer sauber halten und (automatisch) formatieren
11. Open/Closed Principle (OC)
  - Klassen und Methoden sollten offen für Erweiterungen und geschlossen für Änderungen sein
  - Änderung: Anpassung bestehenden Quellcodes (fehleranfällig, da bereits in Verwendung)
    - Änderung einer Schnittstelle → viele Anpassungen im bestehenden Code

- Erweiterung: Implementierung einer Schnittstelle, Hinzufügen einer Methode
  - → Kein Einfluss auf bestehenden Code

12. Interface Segregation Principle (ISP)
    - Aufteilung grosser Schnittstellen in kleinere Schnittstellen sobald Implementierungen unnötige Methoden implementieren müssen
13. Law of Demeter (LoD)
    - Auch bekannt als *«Principle of Least Knowledge»*
      - Objekte sollten nur mit Objekten aus ihrer unmittelbaren Umgebung kommunizieren
      - *«Talk to friends, not to strangers.»*
    - Eine Methode `m` des Objektes `O`sollte nur folgendes verwenden:
      - `O` selber
      - die Eigenschaften von `O`
      - die eigenen Parameter von `m`
      - von `m` selber erstellte Instanzen
      - globale Variablen
14. Liskov Substitution Principle (LSP)
    - Code, der Instanzen einer Basisklasse verwendet, muss auch mit Instanzen davon abgeleiteter Klassen funktionieren, ohne den Code verändern zu müssen. Beispiel:
      - Basisklasse `Person`
      - Unterklasse `Student`
      - `insert(Person p);` muss auch für `insert(Student s);` funktionieren, ohne die `insert`-Methode anpassen zu müssen
    - Beispiel für eine Verletzung des Prinzips:
      - Die Klasse `Kreis` erbt von der Klasse `Ellipse` (ein Kreis ist tatsächlich eine Ellipse!)
      - Die Methode `skaliereX()` und `skaliereY()` funktionieren für Ellipsen, die in beide Achsen skaliert werden können
      - Ein Kreis darf jedoch nur in beide Achsen gleichzeitig skaliert werden!
      - → Das Prinzip hängt weniger von der Klassenstruktur als vom Client-Code ab