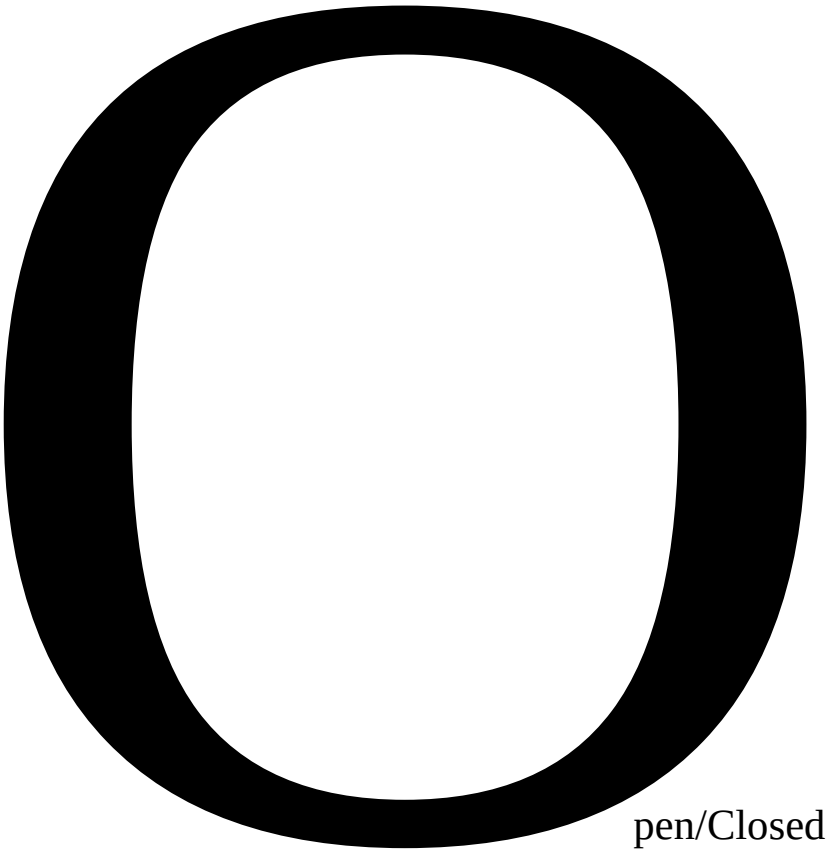
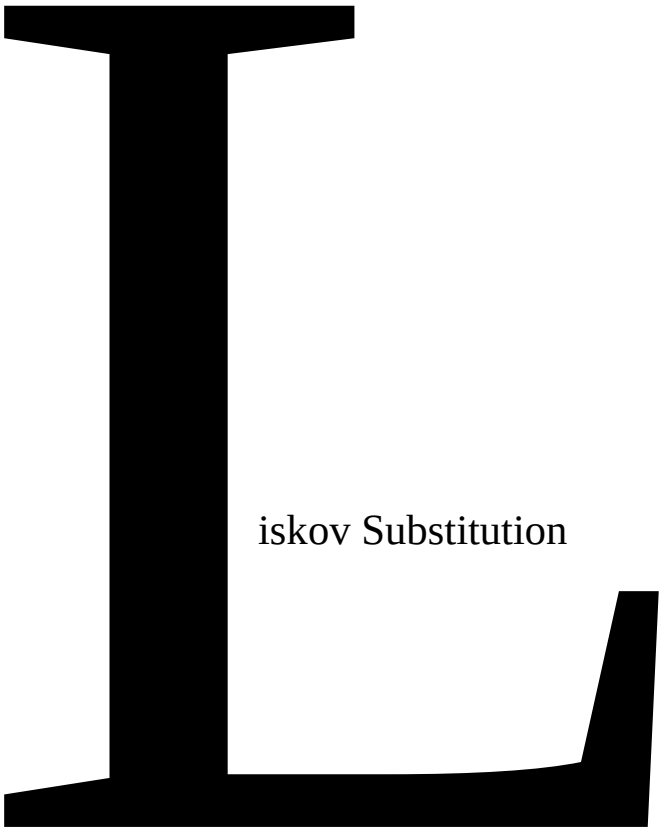
# Single Responsibility

Jede Klasse sollte nur eine einzige Verantwortung haben. Das bedeutet, dass eine Klasse nur einen Grund haben sollte, sich zu ändern. Dies fördert die Kohäsion innerhalb der Klasse und macht sie wartbarer.

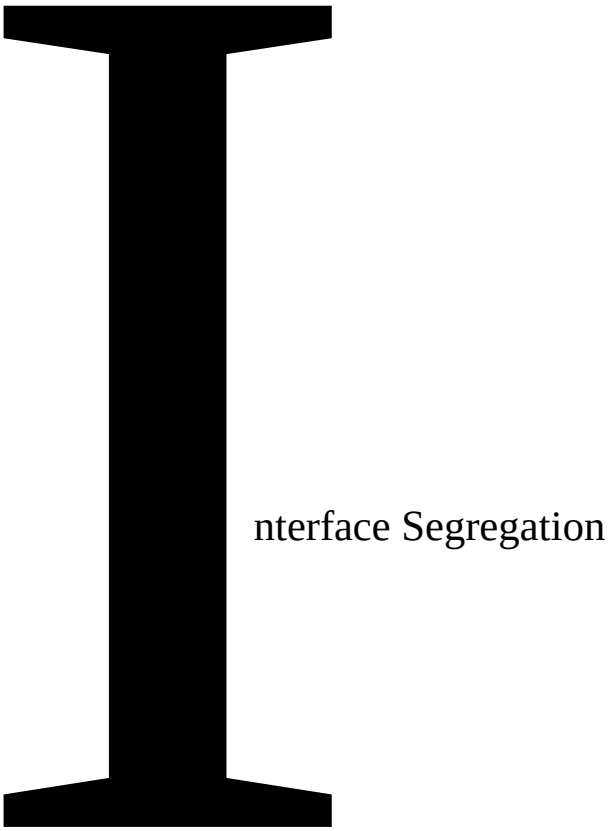
# O




Software-Entitäten (Klassen, Module, Funktionen usw.) sollten für Erweiterungen offen, aber für Modifikationen geschlossen sein. Das bedeutet, dass es möglich sein sollte, das Verhalten einer Entität zu erweitern, ohne den vorhandenen Code zu ändern. Dies wird oft durch die Verwendung von Abstraktion und Polymorphismus erreicht.




Objekte einer Basisklasse sollten in der Lage sein, durch Objekte von abgeleiteten Klassen ersetzt zu werden, ohne dass dies die Korrektheit des Programms beeinträchtigt. Dieses Prinzip betont die Bedeutung der korrekten Erbschaft, sodass eine Unterklasse tatsächlich als Typ ihrer Basisklasse fungieren kann.




Kein Client sollte gezwungen sein, von einem Interface abhängig zu sein, das er nicht verwendet. Das bedeutet, dass größere Interfaces in kleinere, spezifischere Teile aufgeteilt werden sollten, sodass die Implementierenden Klassen nur die Methoden implementieren müssen, die sie tatsächlich benötigen.

# D

ependency Inversion

Abhängigkeiten innerhalb des Codes sollten von Abstraktionen und nicht von Konkretisierungen abhängen. Das heißt, hochrangige Module sollten nicht von niedrigrangigen Modulen abhängen, sondern beide sollten von Abstraktionen abhängen. Dieses Prinzip fördert die Entkopplung der Komponenten und erleichtert das Testen und Warten.

Die SOLID-Prinzipien sind grundlegende Design-Prinzipien in der Softwareentwicklung, die dazu beitragen, wartbaren, erweiterbaren und leicht verständlichen Code zu schreiben. Jeder Buchstabe in SOLID steht für ein eigenes Prinzip:

1. **S - Single Responsibility Principle (SRP)**: Jede Klasse sollte nur eine einzige Verantwortung haben. Das bedeutet, dass eine Klasse nur einen Grund haben sollte, sich zu ändern. Dies fördert die Kohäsion innerhalb der Klasse und macht sie wartbarer.
2. **O - Open/Closed Principle (OCP)**: Software-Entitäten (Klassen, Module, Funktionen usw.) sollten für Erweiterungen offen, aber für Modifikationen geschlossen sein. Das bedeutet, dass es möglich sein sollte, das Verhalten einer Entität zu erweitern, ohne den vorhandenen Code zu ändern. Dies wird oft durch die Verwendung von Abstraktion und Polymorphismus erreicht.
3. **L - Liskov Substitution Principle (LSP)**: Objekte einer Basisklasse sollten in der Lage sein, durch Objekte von abgeleiteten Klassen ersetzt zu werden, ohne dass dies die Korrektheit des Programms beeinträchtigt. Dieses Prinzip betont die Bedeutung der korrekten Erbschaft, sodass eine Unterklasse tatsächlich als Typ ihrer Basisklasse fungieren kann.
4. **I - Interface Segregation Principle (ISP)**: Kein Client sollte gezwungen sein, von einem Interface abhängig zu sein, das er nicht verwendet. Das bedeutet, dass größere Interfaces in kleinere, spezifischere Teile aufgeteilt werden sollten, sodass die Implementierenden Klassen nur die Methoden implementieren müssen, die sie tatsächlich benötigen.
5. **D - Dependency Inversion Principle (DIP)**: Abhängigkeiten innerhalb des Codes sollten von Abstraktionen und nicht von Konkretisierungen abhängen. Das heißt, hochrangige Module sollten nicht von niedrigrangigen Modulen abhängen, sondern beide sollten von Abstraktionen abhängen. Dieses Prinzip fördert die Entkopplung der Komponenten und erleichtert das Testen und Warten.