Programmierprinzipien

Dieser Artikel erfüllt die GlossarWiki-Qualitätsanforderungen **nur teilweise**:

Korrektheit: 5 (vollständig überprüft)	**Umfang**: 4 (unwichtige Fakten fehlen)	**Quellenangaben**: 2 (wichtige Quellen fehlen)	**Quellenarten**: 3 (gut)	**Konformität**: 5 (ausgezeichnet)

Anmerkung: In diesem Artikel haben die beiden Folgenrungspfeile „→“ und „⇒“ folgende Bedeutungen:

⇒: „hat zur Folge“

→: „trägt bei zu“, „verbessert“, „erhöht“

Inhaltsverzeichnis

1 Zweck

Um nützliche und dennoch preiswerte Programme und Anwendungen erstellen zu können, sollte man eine Reihe von Programmierprinzipien beachten.

Bevor die Programmierprinzipen näher beschrieben werden können, werden zunächst die Ziele genauer spezifiziert, die mit diesen Prinzipien erreicht werden sollen.

2 Ziele

Ziel eines jeden Softwareentwicklungs-Vorhaben sollte die Erstellung von Software sein, die zwei Eigenschaften aufweist:

nützlich (useful)
: Die Software sollte für den Benutzer nützlich sein.

preiswert (inexpensive)
: Die Software sollte so geringe Kosten und Folgekosten verursachen wie möglich.

3 Software-Eigenschaften zur Erfüllung der beiden Ziele

Um die beiden vorgenannten Ziele zu erreichen, sollte ein Softwarepaket folgende Eigenschaften haben:

spezifiziert (specified)
: Um die Nützlichkeit und die Korrektheit des Softwarepakets beurteilen zu können, bevor es eingesetzt wird, sollte eine Spezifikation vorliegen, die detailliert die Aufgaben und Eigenschaften des Paketes beschreibt.
⇒ Nützlichkeit (ist abschätzbar)

korrekt (correct)
: Die Implementierung des Paketes sollte korrekt sein, es sollte also die Spezifikation in allen Punkten erfüllen.
⇒ Nützlichkeit: Eine Software, die nicht korrekt arbeitet, nutzt dem Benutzer in den fehlerhaften Situationen nicht.
⇒ keine Kosten durch falsches oder unerwartetes Verhalten

robust (robust)
: Das Softwarepaket sollte auch in abnormalen (nicht-spezifizierten) Situationen stabil laufen.

⇒ keine Kosten durch Inkonsistenzen

⇒ geringe Kosten bei Bedienungsfehlern und Systemausfällen

benutzbar (usable)

Das Softwarepaket sollte einfach zu erlernen und zu benutzen sein.

⇒ Nützlichkeit wird verbessert

⇒ geringe Schulungskosten

⇒ fehlerhafte Nutzung kommt nur selten vor und hat daher nur selten Kosten zu Folge

⇒ keine Kosten durch frustierte Benutzer

sicher (secure)

Unautorisierter Zugriff auf Daten oder Programme sollte unmöglich sein.

⇒ keine Kosten durch Sicherheitsmängel

effizient (efficient)

Das Softwarepakt sollte eine möglichst gute Laufzeit- und Speichereffizienz aufweisen, d.h. die vorhandenen Ressourcen möglichst gut ausnutzen.

⇒ geringe Personalkosten durch geringe Wartezeiten

⇒ geringe Kosten durch geringe Hardwareanforderungen

wartbar (maintainable)

Das Softwarepaket sollte leicht an neue Gegebenheiten angepasst werden können. Fehler sollten leicht behoben werden können (da sie sich nie ganz vermeiden lassen).

⇒ geringe Wartungskosten

Für die Eigenschaft „wartbar“ gibt es drei Spezialfälle, die hier gesondert genannt werden sollen:

kompatibel (compatible)

Das Softwarepaket sollte komaptibel zu bestehenden Systemen sein, es sollte also standardisierte Schnittstellen unterstützen.

⇒ geringe Intergrations- und Anpassungskosten

portabel (portable)

Das Softwarepaket sollte einfach auf neue Systeme portiert werden können.

⇒ geringe Kosten bei Änderung der Ablaufumgebung

erweiterbar (extensible)

Erweiterungen der Spezifikation sollten schnell implementiert werden können.

⇒ geringe Wartungskosten

4 Programmierprinzipien

Um die vorgenannten Ziele zu erreichen, sollten folgende Programmierprinzipien beachtet werden:

4.1 Verständlichkeit, Comprehesibility, Lesbarkeit, Readability

Schreibe lesbaren und verständlichen Code.

Der Programmcode sollte so einfach wie möglich gelesen und verstanden werden können.

Das heißt, der Code sollte sauber formartiert werden, es sollten sprechende Bezeichner verwendet werden, es sollten sinnvolle Kommentare eingefügt werden etc. Ein erfahrender Programmierer sollte die Bedeutung der einzelnen Anweisungen, Operationen, Defintionen etc. problemlos erfassen können.

→ Überprüfbarkeit (durch Programmierer und nicht nur durch Compiler u.Ä.) (→ Korrektheit, Robustheit)

→ Wartbarkeit (da die Programme auch von anderen Programmierern verstanden werden)

→ Erweiterbarkeit (da die Programme auch von anderen Programmierern verstanden werden)

4.2 Schreibbarkeit, Writability

Verwende eine Programmiersprache und -umgebung, die dich beim Schreiben des Codes unterstützt.

Der Programmcode sollte so einfach wie möglich geschrieben werden können. Zum einen sollte es die Programmiersprache ermöglichen, einfachen und eleganten Code zu schreiben. Und zum anderen sollten die Programmwerkzeuge einen bei der Erstellung des Codes möglichst gut unterstützen.

Vorteile:

Code kann schneller erstellt werden.
Code kann automatisch formatiert werden.
Fehler/Warnungen werden von der Programmierumgebung zeitnah gemeldet.
Fehler werden vermieden (wenn beispielsweise Code-Fragmente automatisch generiert werden).

→ Korrektheit

→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

4.3 Stetigkeit, Continuity

Schreibe stetigen Code.

Schreibe den Code so, dass kleine Änderungen an der Spezifikation auch nur kleine Änderungen am Code zur Folge haben.

→ Wartbarkeit (der Code kann schnell an neue Gegebenheiten angepasst werden)

→ Erweiterberkeit (der Code kann schnell an neue Gegebenheiten angepasst werden)

→ Wiederverwendbarkeit (der Code kann schnell an neue Gegebenheiten angepasst werden)

4.4 Konfigurierbarkeit, Customizability

Schreibe konfigurierbaren Code.

Konstante Werte sollten im Allgemeinen nicht direkt in den Code eingefügt werden, sondern als konstante Werte separat definiert werden (Ausnahme: triviale Konstanten, die sich sicher nie ändern werden, wie z.B. Vergleiche mit dem Wert). Konstanten, die das Programmverhalten beeinflussen, sollten im Allgemeinen beim Programmstart aus einer Konfigurationsdatei ausgelesen werden.

→ Stetigkeit (→ Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit)

4.5 Don't repeat yourself, DRY[1]

Wiederhole dich nicht.

Code sollte nicht dupliziert und anschließend aber gar nicht oder nur marginal modifiziert werden.

→ Stetigkeit (→ Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit)

→ Verständlichkeit (da weniger Code existiert) (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

→ Robustheit (da keine inkonsitenten Änderungen an mehreren Stellen möglich sind)

4.6 Repeat yourself, RY[2]

Wiederhole dich.

In Benutzerschnittstellen sollten gleiche oder vergleichbare Aufgaben immer auf dieselbe Weise vom Benutzer durchgeführt werden können. Es sollten dieselben Bezeichner, dieselben bzw. gleichartige Bedienelemente, dasselbe Design etc. verwendet werden.

→ Wiederverwendbarkeit (da gleichartige Elemente nicht mehrfach implementiert werden müssen; DRY)

→ Stetigkeit (→ Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit)

→ Verständlichkeit (da weniger Code existiert) (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

4.7 Gesetz von Demeter[3], Law of Demeter, LoD

„Sprich nur zu deinen nächsten Freunden.“

Ein Objekt sollte nur Methoden von Objekten aufrufen, die es „persönlich“ kennt:

Objekte, die in Zustandsvariablen gespeichert sind
Objekte, die über direkte Beziehungen zugänglich sind
Objekte, die mittels Parametern beim Methodenaufruf übergeben wurden
Objekte, die das aktuelle Objekt selbst erzeugt hat

→ Stetigkeit (→ Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit)

→ Verständlichkeit (da die Anzahl der Kommunikationspartner geringer ist) (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

4.8 Überprüfbarkeit, Verifiability

Schreibe überprüfbaren Code.

Die Korrektheit eines Softwarepakets sollte plattformunabhängig sein und automatisch überprüft werden können. Viele Programmierparadigmen und auch Projekttechniken verlangen zu diesem Zweck, dass gleichzeitig mit der Software auch passende Softwaretests implementiert werden (Modultests, Unittests). Noch besser ist es, wenn die Semantik des Programmes formal spezifiert und überprüft werden kann (Abstrakte Datentypen, Integritätsbedingungen, ...)

→ Korrektheit (insbesondere, wenn die Erfüllung der Spezifikation verifiziert, d.h. formal nachgewisen wird)
→ Robustheit (wenn die Überprüfungen auch durchgeführt werden)
→ Portierbarkeit (wegen der Plattformunabhängigkeit)

4.9 Benutze Schnittstellen, Make Use of Interfaces

Benutze Schnittstellen zur Kommunikation mit Objekten, Modulen etc.

Die Verwendung von Schnittstellen zum Zugriff auf und zur Modifikation von Daten/Informationen (an Stelle eines direkten Zugriffes) ermöglicht es, Code lokal zu ändern, ohne dass dies Auswirkungen auf andere Objekte, Module etc. hat (solange sich die Schnittstellen nicht ändern).
→ Spezifikation
→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)
→ Stetigkeit (→ Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit)
→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)
→ Modularität

4.10 Benutze Integritätsbedingungen, Make Use of Integrity Constraints, Design by Contract[4]

Benutze und beachte Integritätsbedingungen.

Wann immer möglich sollten Integritätsbedingungen – Vorbedingungen (pre conditions), Nachbedingungen (post conditions), Invarianten (invariants), Zusicherungen (Assertions), Typdefinitionen etc. – definiert und deren Einhaltung auch überprüft werden.
→ Spezifikation
→ Korrektheit
→ Robustheit
→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)

4.11 Liskovsches Substitutionsprinzip[5], LSP, Ersetzbarkeitsprinzip, Liskov substitution principle[6]

Überschreibe Methoden nicht mit unerwartetem Code.

Eine Methode sollte nicht so überschrieben werden, dass sich ein Objekt einer abgeleiteten Klasse überraschend anders verhält, als man es aufgrund der Definition der Basisklasse erwarten würde. Mit anderen Worten: Methoden, die in abgeleiteten Klassen neu definiert werden, müssen alle Integritätsbedingungen (d.h. die Spezifikation) der Basisklasse beachten. Dieses Prinzip kann also ein Spezialfall des vorangehenden Prinzips angesehen

werden.

→ Spezifikation

→ Korrektheit

→ Robustheit

4.12 Modularität, Modularity, Teile und herrsche, Divide et impera

Teile und herrsche

Ein komplexes Problem (Vorhaben, Projekt, ...) kann man nur dadurch in den Griff bekommen, dass man es solange in kleinere, möglichst unabhängige Teilprobleme (Teilvorhaben, Phasen, Vorgänge, ...) zerlegt, bis diese lösbar (= beherrschbar) sind. Ein Programm sollte daher **modularisiert**, d.h. in einzelne **Module** unterteilt werden. Jedes Modul sollte nur für eine Aufgabe zuständig sein und diese korrekt und robust erfüllen.

4.12.1 Fünf Anforderungen von Bertrand Meyer

Bertrand Meyer war einer der ersten, der die Bedeutung der Modularität erkannt und beschrieben hat. Er hat insgesamt fünf Anforderungen formuliert, die bei der Definition von Modulen beachtet werden sollten:[4]

4.12.1.1 Modulare Zerlegbarkeit, modular decomposability

Zerlege ein Problem in unterschiedliche, seperat lösbare Teilprobleme.

→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

→ Wiederverwendbarkeit (wenn die Module weitgehend unabhängig voneinander sind)

→ Erweiterbarkeit

4.12.1.2 Modulare Zusammenfügbarkeit, modular composability

Module sollten in möglichst vielen (unterschiedlichen) Situationen/Anwendungen eingesetzt werden können.

Das heißt, Module sollten auf unterschiedliche Art und Weise zu neuen Funktionseinheiten zusammengefügt werden können.

→ Wiederverwendbarkeit

→ Erweiterbarkeit

→ Wartbarkeit

→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)

4.12.1.3 Modulare Verständlichkeit, modular understandability

Die Aufgaben eines Moduls sollten verstanden werden, ohne dass man viele andere Module kennen muss.

→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

Dies ist ein Spezialfall des Prinzips „Verständlichkeit“.

4.12.1.4 Modulare Stetigkeit, modular continuity

Kleine Änderungen an den Anforderungen sollten nur Änderungen an einer geringen Zahl von Modulen zur Folgen haben.

Im besten Fall ist nur ein einziges Modul betroffen.

→ Stetigkeit (→ Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit)

Dies ist ein Spezialfall des Prinzips „Stetigkeit“.

4.12.1.5 Modulare Robustheit, modular protection

Ein Modul sollte alle abnormalen Fälle selbst behandeln.

Das heißt, es sollte sich nicht darauf verlassen, dass ein anderes Modul die Schnittstellen nur gemäß der Spezifikation verwendet.

→ Korrektheit

→ Robustheit

→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)

→ Wartbarkeit

4.12.2 Sechs Prinzipien von Bertrand Meyer

Aus diesen Forderungen leitet Bertrand Meyer sechs Prinzipien ab, die bei der Entwicklung von Modulen beachtet werden sollten.

4.12.2.1 Syntaktische Einheiten, Syntactical Units

Module sind syntaktische Einheiten.

Module sollten unabhängig voneinander übersetzt und in Bibliotheken zur Verfügung gestellt werden können.

→ Wiederverwendbarkeit

→ Korrektheit

→ Robustheit

→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)

→ Wartbarkeit

4.12.2.2 Wenige Schnittstellen, Few Interfaces

Module sollten wenige Schnittstellen haben.

Module sollten mit so wenig anderen Modulen kommunizieren wie möglich.

→ Wiederverwendbarkeit

→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

4.12.2.3 Schlanke Interfaces, Small Interfaces

Module sollten schlanke Schnittstellen haben.

Schnittstellen sollten nicht mit Funktionalität überfrachtet sein. Das heißt, zwei Module sollten über möglichst umfangarme Schnittstellen miteinander kommunizieren.

→ Wartbarkeit

→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)

4.12.2.4 Explizite Schnittstellen, Explicit Interfaces

Module sollten über explizite Schnittstellen kommunizieren.

Ein Modul sollte nicht über globale Variablen oder andere implizite Kommunikationswege mit anderen Modulen kommunizieren, sondern über explizite Schnittstellen.

→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

4.12.2.5 (Daten-)Kapselung, Geheimnisprinzip, Encapsulation, Information Hiding

Informationen sollten in Modulen gekapselt sein.

Jede Information, die nicht notwendig ist, um die Aufgaben bzw. die Funktionalität eines Moduls zu verstehen bzw. zu benutzen, sollte von außen nicht zugänglich sein.

→ Korrektheit

→ Robustheit

→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)

→ Wiederverwendbarkeit

→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

4.12.2.6 Offen-geschlossen-Prinzip, Open-Closed Principle

Module sollten sowohl offen, als auch geschlossen sein.

Module sollten sowohl offen für Veränderungen, als auch abgeschlossen gegenüber Veränderungen sein. Da heißt, ein funktionierendes Modul sollte möglichst nicht mehr verändert werden, aber andererseits sollte es problemlos möglich sein, ein Modul an neue Gegebenheiten anzupassen. In objektorientierten Sprachen wird dies vor allem durch Vererbung, Überschreiben von Methoden und die Verwendung von Abstrakten Klassen (insbesondere Interfaces) erreicht.

→ Wiederverwendbarkeit (wenn Module offen sind)

→ Erweiterbar (wenn Module offen sind)

→ Robustheit (wenn getestete Module geschlossen sind)

→ Wartbarkeit (wenn funtionierende Module geschlossen sind; jede Änderung an einem Modul hat eventuell weitere Änderungen zu Folgen → dies soll nach Möglichkeit vermieden werden)

4.12.3 Weitere Modul-Prinzipien

Seit den wegweisenden Arbeiten von Bertrand Meyer aus den späten 80er-Jahren wurden weitere Prinzipien entwickelt, die die Definition von Modulen betreffen.

4.12.3.1 Verwende Design Patterns[7]

Verwende die in Entwurfsmustern (design patterns) definierten Modul-„Schablonen“.

In der Programmierung gibt es viele Aufgaben, die immer wiederkehren, wie zum Beispiel das Durchlaufen einer Menge von Objekten. Für zahlreiche derartige Aufgaben gibt es praxiserprobte Modul-„Schablonen“. Diese beschreiben, welche Schnittstellen, Klassen, Objekte etc. benötigt werden, um die jeweilige Aufgabe zu lösen.

→ Wiederverwendbarkeit

→ Korrektheit

→ Robustheit

→ Verständlichkeit (→ Korrektheit, Robustheit, Wartbarkeit, Erweiterbarkeit)

4.12.3.2 Separation of Concerns[8]

Jede Aufgabe wird von genau einem Modul erfüllt.

Man also die „Verantwortlichkeiten“ unter den Modulen aufteilen, sonst gehen die Vorteile der Modularisierung zum Teil wieder verloren.

4.12.3.3 Single Responsibility Principle[9]

Jede Aufgabe wird von genau einem Modul erfüllt und jedes Modul erfüllt genau eine Aufgabe.

Bei der objektorientierten Programmierung treibt man die Modularisierung manchmal sogar

so weit, dass nicht nur für jede Aufgabe höchstens ein Objekt zuständig ist (Separations of Conerns), sondern dass umgekehrt auch jedes Objekt für genau eine Aufgabe zuständig ist (Single responsibility principle).

Das heißt, für jedes Modul gibt es genau einen Grund, warum es geändert werden muss: Die eine Aufgabe, die es hat, ändert sich.

→ Robustheit

4.12.3.4 Interface-Segregation-Prinzip[10], Interface Segregation Principle

Zu große Interfaces sollten in mehrere Interfaces aufgeteilt werden.

Ein Modul, das ein Interface benutzt, sollte nur diejenigen Methoden präsentiert bekommen, die sie auch wirklich benötigt. Dieses Prinzip formuliert eine konkrete Möglichkeit, wie man die von Meyer geforderten schlanken Schnittstellen realisieren kann.

→ Wartbarkeit

→ Wiederverwendbarkeit

→ Überprüfbarkeit (→ Korrektheit, Robustheit, Portierbarkeit)

5 Quellen

1. **Hunt, Thomas (2003)**: Andrew Hunt und David Thomas; Der Pragmatische Programmierer; Verlag: Fachbuchverlag Leipzig im Carl Hanser Verlag; ISBN: 3446223096, 978-3446223097; 2003; Quellengüte: 5 (Buch)
2. **Kowarschick (MMProg)**: Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)
3. **Lieberherr, Holland (1989)**: Karl J. Lieberherr und Ian M. Holland; Assuring Good Style for Object-Oriented Programs; in: IEEE Software; Band: 6; Nummer: 5; Seite(n): 36–48; Verlag: IEEE Computer Society Press; Adresse: Los Alamitos, CA, USA; 1989; Quellengüte: 5 (Artikel)
4. **Meyer (1997)**: Bertrand Meyer; Object-oriented Software Construction; Auflage: 2; Verlag: Prentice Hall International; ISBN: 0136291554; 1997; Quellengüte: 5 (Buch)
5. **Liskov, Wing (1993)**: Barbara Liskov und Jeannette M. Wing; Family Values – A Behavioral Notion of Subtyping; Hochschule: Carnegie Mellon University; Adresse: Pittsburgh, PA, USA; Web-Link; 1993; Quellengüte: 5 (Technischer Bericht)
6. **WikipediaEN:Liskov substitution principle**: Wikipedia-Autoren; Wikipedia, die freie Enzyklopädie – Liskov substitution principle; Organisation: Wikimedia Foundation Inc.; Adresse: San Francisco; http://en.wikipedia.org/w/index.php?title=Liskov_substitution_principle&oldid=499629316; 2012; Quellengüte: 5 (Web)
7. **Gamma et al. (1995)**: Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides; Design Patterns – Elements of Reusable Object-Oriented Software; Auflage: 1; Verlag: Addison-Wesley Longman; Adresse: Amsterdam; ISBN: 0201633612; 1995; Quellengüte: 5 (Buch)
8. **WikipediaEN:Separation_of_concerns**: Wikipedia-Autoren; Wikipedia, die freie Enzyklopädie – Separation_of_concerns; Organisation: Wikimedia Foundation Inc.; Adresse: San Francisco; http://en.wikipedia.org/w/index.php?title=Separation_of_concerns&oldid=508337060; 2012; Quellengüte: 5 (Web)
9. **WikipediaEN:Single responsibility principle**: Wikipedia-Autoren; Wikipedia, die freie Enzyklopädie – Single responsibility principle; Organisation: Wikimedia Foundation Inc.; Adresse: San Francisco; http://en.wikipedia.org/w/index.php?title=Single_responsibility_principle&oldid=507275547; 2012; Quellengüte: 5 (Web)
10. **Martin (1996)**: Robert C. Martin; The Interface Segregation Principle; in: C++ Report; Band: 8; Web-

Link; 1996; Quellengüte: 5 (Artikel)
1. **Wintersteiger, Mathis (2011)**: Andreas Wintersteiger und Christoph Mathis; Clean Code – Prinzipien bei der Entwicklung von sauberem Code; in: Entwickler Magazin; Band: 2011; Nummer: 5; Seite(n): 13–20; Adresse: Frankfurt am Main; ISSN: 1619-7941; 2011; Quellengüte: 5 (Artikel)
2. **Kowarschick (MMProg)**: Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)

6 Siehe auch

1. Softwaretechnik
2. Wikipedia (EN): DRY (Web)
3. WikipediaEn:Category:Programmingprinciples
4. Wikipedia:Prinzipien objektorientierten Designs
5. WikipediaEn:Code smell

TO BE DONE

Principle of Least Surprise

Kategorien:
Programmierprinzip
HowTo