7 Architekturprinzipien

Prinzipien sind Grundsätze, die man seinem Handeln zugrunde legt. Prinzipien sind allgemeingültig, abstrakt, allgemeinster Art. Sie bilden eine theoretische Grundlage. Prinzipien werden aus der Erfahrung und Erkenntnis hergeleitet und durch sie bestätigt.

Definition

In dem »Lehrbuch der Softwaretechnik – Basiskonzepte und Requirements Engineering« [Balz09a, S. 25 ff.] werden folgende allgemeinen Prinzipien vorgestellt, die für die Softwaretechnik relevant sind:

- Prinzip der Abstraktion
- Prinzip der Strukturierung
- Prinzip der Bindung und Kopplung
- Prinzip der Hierarchisierung
- Prinzip der Modularisierung
- Geheimnisprinzip
- Prinzip der Lokalität
- Prinzip der Verbalisierung

Alle diese Prinzipien sind auch für den Softwareentwurf relevant, in Abhängigkeit von den funktionalen und nichtfunktionalen Anforderungen in unterschiedlicher Gewichtung und Ausprägung.

In der Literatur werden noch weitere spezifische Architekturprinzipien diskutiert, oft auch unter anderen Bezeichnungen wie Leitprinzipien, Qualitätsindikatoren, Hinweise *(considerations)*. Bisweilen wird auch von »architektonischer Schönheit« gesprochen. Eine Reihe dieser Architekturprinzipien wird näher vorgestellt:

- »Architekturprinzip: Konzeptionelle Integrität«, S. 30
- »Architekturprinzip: Trennung von Zuständigkeiten«, S. 31
- »Architekturprinzip: Ökonomie«, S. 32
- »Architekturprinzip: Symmetrie«, S. 33
- »Architekturprinzip: Sichtbarkeit«, S. 34
- »Architekturprinzip: Selbstorganisation«, S. 34

Die aufgeführten Prinzipien können durch verschiedene Maßnahmen erreicht werden, beispielsweise durch den Einsatz geeigneter Architektur- und Entwurfsmuster (siehe »Architektur- und Entwurfsmuster«, S. 37).

H. Balzert, *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb,*
DOI 10.1007/978-3-8274-2246-0_7, © Spektrum Akademischer Verlag Heidelberg 2011

7.1 Architekturprinzip: Konzeptionelle Integrität

Frage Was stellen Sie sich unter dem Architekturprinzip »Konzeptionelle Integrität« vor?

Antwort Eine Softwarearchitektur besitzt *keine* »Konzeptionelle Integrität«, wenn für gleichartige Aufgabenstellungen unterschiedliche Architekturansätze, z. B. unterschiedliche Entwurfsmuster oder Variationen davon, verwendet werden. Das führt zu einer schlechten Wartbarkeit und Verständlichkeit.

Das **Architekturprinzip** der **Konzeptionellen Integrität** *(conceptual integrity)* ist eingehalten, wenn Entwurfsentscheidungen im gesamten System durchgängig angewendet und Speziallösungen vermieden werden.

Beispiel
- ○ Entwickler A verwendet für die Fehlerbehandlung das Konzept der Ausnahmebehandlung.
- ○ Entwickler B verwendet für Fehlermeldungen Rückgabewerte.
- ○ Entwickler C protokolliert Fehler lediglich in einem Logbuch.

Beispiel Das Architekturprinzip wird verletzt, wenn in einer verteilten Architektur für die Komponenten mehrere verschiedene Schnittstellen und Implementierungen für die Kommunikation angeboten werden.

Die Beachtung dieses Architekturprinzips bringt folgende Vorteile mit sich:
- ✚ Das Verständnis der Architektur wird durch die konzeptionelle Integrität erleichtert.
- ✚ Durch die Vermeidung von Speziallösungen und punktuellen Ausnahmen wird die Komplexität der Architektur reduziert.

Die Einhaltung dieses Prinzips wird bei einer inkrementellen Entwicklung einer Architektur schwierig, da das Entstehen einer Architektur über einen längeren Zeitraum oft zu Stilbrüchen führt. Bei Änderungen einer Architektur wird oft nicht die gesamte Architektur bearbeitet, sondern nur Teile, was Integritätsbrüche zur Folge haben kann. Ebenso kommt es bei der Implementierung einer Architektur zur Verletzung der Integrität, wenn Entwickler wegen besonderer Anforderungen, Optimierungsanforderungen oder fehlendem Problembewusstsein Speziallösungen erfinden.

Terminologie Synonym zu dem Begriff »Konzeptionelle Integrität« findet man in der Literatur oft noch folgende Begriffe: Prinzip der kleinstmöglichen Überraschung, Entwurfssymmetrie, Einheitlichkeit.

Literatur [Stal09, S. 141], [ReHa09, S. 89 f.], [PBG07, S. 108]

7.2 Architekturprinzip: Trennung von Zuständigkeiten

Frage

Was können Sie sich unter dem Architekturprinzip »Trennung von Zuständigkeiten« vorstellen?

Antwort

Jede Komponente einer Architektur ist bei dem Prinzip **Trennung von Zuständigkeiten** *(separation of concerns)* für eine Aufgabe oder einen Aufgabenkomplex verantwortlich. Die Aufgabe ist *nicht* über mehrere Komponenten verteilt. Eine Komponente ist also verantwortlich für eine einzige Aufgabe und erledigt diese Aufgabe richtig. Ist eine Komponente für einen Aufgabenkomplex zuständig, dann soll die Komponente für jede von ihr zu erledigende Aufgabe eine eigene Schnittstelle zur Verfügung stellen.

Beispiel

Die Abb. 7.2-1 zeigt zwei verschiedene Möglichkeiten, Zuständigkeiten zu trennen [BuHe10a, S. 65]. Auf der linken Seite werden Zuständigkeiten durch eine Vererbungshierarchie abgebildet, die zu überfrachteten Blatt-Klassen führen können. Die Wurzel-Klasse stellt die Infrastruktur zur Verfügung, Unterklassen sorgen für die Anpassung an die aktuellen Domänen-Unterklassen. Diese Art der Trennung von Zuständigkeiten erhöht die Abhängigkeiten der Blatt-Klassen.
Auf der rechten Seite der Abb. 7.2-1 ist eine Trennung der Zuständigkeiten durch die Verwendung der Komposition vorgenommen worden (siehe »Schnittstellen, Fabriken und Komposition«, S. 503). Der Fokus liegt auf den Domänen-Klassen. Die Infrastruktur-Klassen befinden sich erst an dritter Stelle. Die Delegation durch konzeptionelle Wurzeln, ausgedrückt durch reine Schnittstellen, führt zu einer strikten und besser separierten Schichtenbildung.

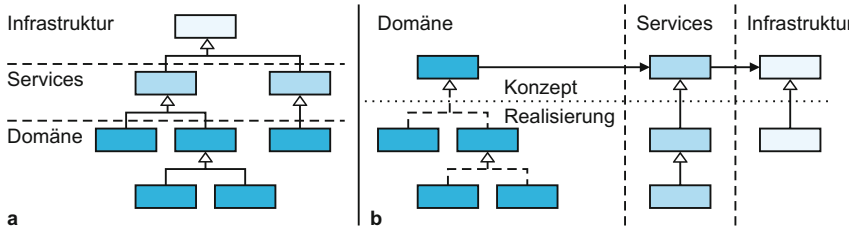

Abb. 7.2-1: Zwei Optionen, um die Trennung von Zuständigkeiten zu realisieren (in Anlehnung an [BuHe10, S. 65]).

Bei der Implementierung wird dieses Prinzip manchmal verletzt, weil aus Effizienzgründen oder wegen der Beschränkung der Programmiersprache mehrere Aspekte im Code einer Komponente realisiert sind, *Tangled Code* genannt.

Die Einhaltung dieses Prinzips bringt folgende Vorteile mit sich:

+ Die Sichtbarkeit wird erhöht, da die Zuständigkeiten klar voneinander abgegrenzt sind.

+ Die Trennung von Zuständigkeiten führt zu einer modularen Architektur.

Die Aufteilung von Zuständigkeiten muss ausgewogen sein. Eine zu starke Trennung von Zuständigkeiten führt zu einer Fragmentierung und einem Verlust an Sichtbarkeit. Durch die Analyse von Gemeinsamkeiten und Unterschieden kann identifiziert werden, was in einer Architektur getrennt werden sollte und was als ganze, kohärente Einheit zu betrachten ist.

Terminologie In der Literatur werden für dieses Prinzip auch die Namen *Spacing* [BuHe10a, S. 65] und »Trennung von Belangen« verwendet [Stal09, S. 143].

Literatur [Stal09, S. 143], [ReHa09, S. 79 f.], [BuHe10a, S. 65]

7.3 Architekturprinzip: Ökonomie

Frage Was könnte das Architekturprinzip »Ökonomie« für eine Softwarearchitektur bedeuten?

Antwort Ein ökonomischer Architekturentwurf sorgt dafür, dass Übersichtlichkeit, Verdichtung und Abstraktion in einem ausgewogenen Verhältnis berücksichtigt werden (in Anlehnung an [BuHe10a, S. 63]). Verdichtung meint Geradlinigkeit und Klarheit im Entwurf.

Übersichtlichkeit dient dazu, ein Durcheinander zu vermeiden. Verdichtung erleichtert das Verständnis der Architektur. Abstraktion sorgt dafür, dass unnötiges Detail verschwindet. Durch dieses Prinzip soll verhindert werden, dass Softwarearchitekturen mit unnötiger Komplexität belastet werden.

Beispiele Beispiele für unnötige Komplexität sind: Willkürliche Flexibilität; unnötige Funktionen; Entwurfsentscheidungen, deren Komplexität nicht problemgerecht ist; Fokus auf Wiederverwendbarkeit statt auf die Benutzbarkeit.

Terminologie Im Zusammenhang mit dem Prinzip Ökonomie werden oft in ähnlicher oder leicht abgewandelter Bedeutung die Prinzipien **Einfachheit** und **Minimalismus** genannt.

Zitat »Je weniger Subsysteme, Komponenten und Interaktionen in einem Softwaresystem auftreten, desto verständlicher die zugehörige Architektur und desto geringer die Gefahr von Fehlern. Unbeabsichtigte Komplexität ergibt sich beispielsweise, sobald Architekten nach Entwurfsperlen streben oder das System auf alle Eventualitäten vorbereiten wollen. Unnötige Indirektionsstufen (accidental complexity) reduzieren in diesem Fall direkt die architektonische Einfachheit« [Stal09, S. 142].

Zu beachten ist allerdings, dass jede Aufgabe eine inhärente Komplexität besitzt. Daher können keine Lösungen mit geringerer Komplexität vorhanden sein.

[BuHe10a, S. 63 f.], [Stal09, S. 142] Literatur

7.4 Architekturprinzip: Symmetrie

Was können Sie sich unter dem Architekturprinzip »Symmetrie« vorstellen? Frage

Symmetrie macht sich durch einen konsistenten, geordneten und ausgewogenen Entwurf bemerkbar. Elemente wiederholen und stützen sich gegenseitig, machen es leichter das Verhalten des Systems vorherzusagen, es zu warten und auszubauen [BuHe10b, S. 12]. Antwort

Es lassen sich verhaltensorientierte und strukturelle Symmetrien unterscheiden.

Beispiele für verhaltensorientierte Symmetrien

- ○ Zu jeder Initialisierungsfunktion soll es auch eine komplementäre Abschlussfunktion geben.
- ○ Zu einem Verbindungsaufbau in einer verteilten Anwendung soll es auch einen Verbindungsabbau im selben Kontext geben, z. B. Herstellung einer Datenbank-Verbindung durch `connect()` und Trennen durch `disconnet()`.
- ○ Jedes erzeugende Entwurfsmuster (siehe »Architektur- und Entwurfsmuster«, S. 37) wird vollständiger und nützlicher, wenn es auch ein Konzept zur »Entsorgung« bzw. »Zerstörung« enthält.
- ○ Jede gestartete Transaktion muss auch beendet werden.

Verhaltensorientierte Symmetrien führen oft zu strukturellen Symmetrien.

Die Erkenntnis, dass in einem erzeugenden Entwurfsmuster auch ein Konzept zur »Entsorgung« bzw. »Zerstörung« fehlt, führt zu einem modifizierten Muster, das strukturell ein Konzept zur »Entsorgung« bzw. »Zerstörung« enthält. Beispiel

Wenn Asymmetrie jedoch zu einem einfacheren Entwurf führt, dann sollte die Symmetrie *nicht* erzwungen werden.

In Java werden Objekte explizit erzeugt. Ihre Zerstörung wird implizit durch den *Garbage Collector* vorgenommen. Beispiel

Die Beachtung des Prinzips der Symmetrie bringt folgende Vorteile mit sich:

- + Ein symmetrischer Entwurf ist leichter zu verstehen, nachzuvollziehen und zu kommunizieren.
- + Die Paarung von »Erzeugung« und »Zerstörung« führt in manchen Fällen zu einer Verbesserung der Architekturqualität.

Symmetriebrüche sind erlaubt, müssen aber gut begründet sein.

Tipp

> Die meisten Softwarearchitekturen benötigen mehr Symmetrie und nicht weniger. Wenn Sie im Zweifel sind, dann machen Sie Ihre Architektur symmetrisch.

Literatur [BuHe10b, S. 12 f.], [Stal09, S. 141]

7.5 Architekturprinzip: Sichtbarkeit

Frage Was stellen Sie sich unter dem Architekturprinzip »Sichtbarkeit« *(visibility)* vor?

Antwort Eine Architektur ist dann gut »sichtbar«, wenn ihre Artefakte deutlich und ausdrucksstark beschrieben sind. Das **Prinzip der Verbalisierung** ist eine Untermenge des Prinzips der Sichtbarkeit. Eine schlechte Sichtbarkeit liegt vor, wenn wichtige Entwurfs- und Fachkonzepte nicht beschrieben oder implizit in der Architektur verborgen sind.

Beispiel Schnittstellen sind oft nicht ausdrucksstark beschrieben und ihre Aufteilung ist unklar. Eine Dokumentation fehlt oder ist ausschweifend. Die Folgen davon sind eine schlechte Modifizierbarkeit und Wartbarkeit. Die Einarbeitung ist aufwändig.

Beispiel Variablen vom Typ `String` repräsentieren oft Konzepte der Domäne, z. B. ISBN-Nummern oder SQL-Anweisungen. Dies ist jedoch nicht die geeignete Repräsentation in einem Programm. Besser ist es, einen expliziten Typ für ein solches Konzept einzuführen. Dadurch werden duplizierter Code und unausgesprochene Entwurfsannahmen vermieden.

Die Verwendung von Mustern macht Konzepte sichtbar.

Das Prinzip der Sichtbarkeit bringt folgenden Vorteil mit sich:

\+ Änderbarkeit, Wartbarkeit und Einarbeitung werden erleichtert.

Literatur [BuHe10a, S. 64 f.]

7.6 Architekturprinzip: Selbstorganisation

Frage Was fällt Ihnen zum Architekturprinzip »Selbstorganisation« ein?

Antwort In vielen Softwaresystemen gibt es einen Trend zu zentralisierten und expliziten Kontroll-, Steuerungs- und Entscheidungsstrukturen. Eine Reihe von Aufgaben lassen sich aber besser über dezentrale Ansätze lösen. Die Kontrolle ist dann nicht einer einzelnen Komponente zugeordnet, sondern die beteiligten Komponenten organisieren sich selbst.

Gibt es für wichtige Funktionen nur eine Server-Instanz, dann wird diese zum Flaschenhals bezogen auf Zuverlässigkeit, Verfügbarkeit und Skalierbarkeit. Beispiel

Das Schreiben von nebenläufigen Programmen, die in mehreren *threads* ablaufen, ist schwierig. Wenn sie nicht sorgfältig entworfen werden, leiden sie unter *race conditions*. Um solche Probleme zu vermeiden, werden häufig übermäßig Synchronisationsmechanismen und eine strikte zentralisierte Laufzeitkontrolle eingesetzt. Als Effekte ergeben sich oft *deadlocks* sowie langsame, instabile und kaum skalierbare Programme – genau das Gegenteil von dem, was man erreichen wollte. Beispiel
Eine dezentralisierte Steuerung ist oft besser, um die Vorteile der Nebenläufigkeit zu nutzen.

Der Entwurf einer Architektur mit sich selbst organisierenden Komponenten erfordert ein tiefes Verständnis der System-Domäne.

Anstelle des Begriffs Selbstorganisation wird oft auch die Bezeichnung »Emergenz« *(emergence)* verwendet. Terminologie

Die Beachtung des Prinzips der Selbstorganisation bringt folgenden Vorteil mit sich:

+ Eine sich selbst organisierende Steuerung ist oft der Schlüssel zu skalierbaren, effizienten und ökonomischen Softwarearchitekturen.

[BuHe10b, S. 13 f.], [Stal09, S. 144] Literatur

http://www.springer.com/978-3-8274-1706-0

Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb

Balzert, H.

2011, XVIII, 596 S. 320 Abb. in Farbe., Hardcover
ISBN: 978-3-8274-1706-0