Entwurf

Es wurde hauptsächlich Material aus (Anja 2020; Helmut 2011) verwendet, wenn nicht anders hingewiesen.

Die Softwarearchitektur besteht aus Architekturbausteinen und ihren Beziehungen und Interaktionen zueinander, sowie ihre physikalische Verteilung. Die externe Sicht eines Bausteins wird durch Schnittstellen beschrieben.

Es gibt verschiedene Sichten auf Architektur:

- Kontext (Use-Case)
- Laufzeit (Timing)
- Verteilung (Komponenten)
- Statik (Klassen)

Bausteine einer Architektur lassen sich untergliedern in:

- Subsysteme
- Komponente
- Frameworks
- Pakete
- Klassen

Subsystem

Subsysteme würden ausschließlich die Komponenten genannt werden, die **nicht** von außen durch den Anwender angesprochen werden können[1]. Der Begriff *Subsystem* wird in der Informatik nicht eindeutig definiert. Ein konkretes Beispiel eines Subsystems ist eine Betriebssystemerweiterung, z.B. in der Windows NT-Familie. Es wird dort zur Verwaltung von bestimmten Prozessarten verwendet und stellt eine API zur Verfügung.[2]

Eigenschaften:

- build a logical unit
- define a scope
- declare coupled functions
- fullfill a part of requirements
- can consist of classes, pakets being logically bound

Das Subsystem ist somit eine Spezialisierung der Komponente, die wiederum eine Spezialisierung einer Klasse ist. Zur besseren Unterscheidung zum Paket: Im Gegensatz zu einem Paket stellt eine Komponente ein physische Sicht dar. Ein Paket dient zur Wahrung der Übersicht großer Systeme oder Subsysteme.

Komponente

Stellt Schnittstellen zur Außenwelt bereit und kann nur durch diese angesprochen werden. Eine Komponente kann durch eine andere Komponente mit

[1] UML2 Glasklar, S.222, Z. 4-5
[2] Wikipedia: Teilsystem

gleichen Schnittstellen ersetzt werden, ohne dass das umgebende Gesammtsystem angepasst werden muss.

Eine Komponente:

- kann beliebig ineinander geschachtelt sein.
- bietet durch die bereitgestellten Schnittstellen einen Zugriffsschutz.
- kann aufgrund der definierten Schnittstellen unabhängig ausgeliefert werden.

Framework

Ein Framework ist ein anpassbares oder erweieterbares System von kooperierenden Klassen, die einen wiederverwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren. Es besteht aus konkreten und abstrakten Klassen, die Schnittstellen definieren.

Pakete *(packages)*

sind Strukturmechanismem, um Klassen und Pakete zu eine Einheit zusammenzufassen. Ein Packet kann selbst Packete enthalten. Packete definieren Namensräume.

Anwendung

Auf der Architekturebene finden sich vor allem:

- Subsysteme
- Komponenten
- Frameworks
- Pakete

als Architekturbausteine wieder. Mit ihnen wird die **Makroarchitektur** gebildet. Sie besitzt ein hohes Abstraktionsnivau.

Makroarchitektur Darunter verstehm man eine sogenannte High-Level-Architektur einer Software, z.B. Model-View-Control [fig. 1].

- *View*

  ist eine Komposition aus GUI-Elementen (Labels, Buttons, Text, etc)

  delegiert die Benutzereingaben zum Controller

- *Controller*

  handhabt und interpretiert die Benutzereingaben

  benachrichtigt das Modell und setzt deren interne States

- *Modell*

  beinhaltet die Logik der Application und verwaltet Daten und States

  benachrichtig das View (,Control), wenn Änderungen auftreten

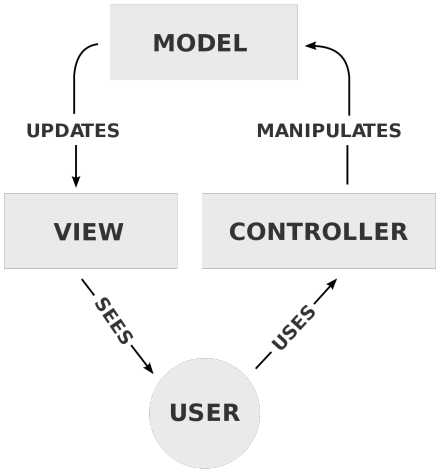

Figure 1: Makroarchitektur: Model-View-Control

Auf nicht ganz so hohem Abstraktionsnivau ist die Mikroarchitektur. Hierfür werden Klassendiagramme verwendet.

Mikroarchitektur Ist die Softwarestruktur eines Artefaktes, z.B. als Low-Level-Modell sogenanntes *Design-Pattern*.

Ein Beispiel für Mikroarchitektur ist das Composite-/ (Kompositum-) /-Pattern [fig. 2].

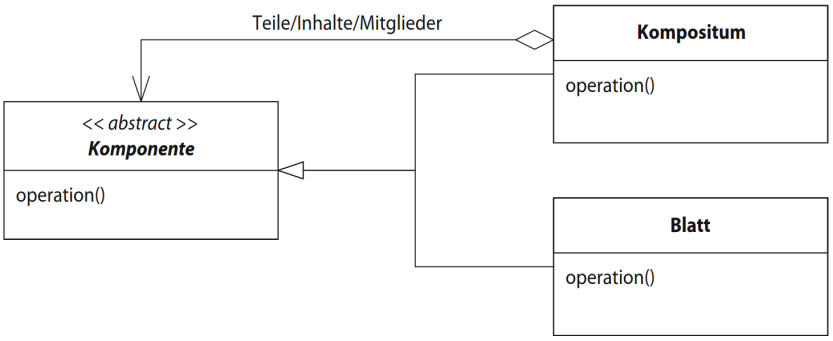

Figure 2: Mikroarchitektur: Composite-Pattern[3]

In [fig. 2], das Klassendiagramm des Composite-Patterns bildet eine Baumstruktur ab mit Blättern und Knoten (Kompositum). Vorteil: Eine Operation muss

[3]Patterns kompakt: Karl Eilebrecht

nur auf das oberste Element angewendet werden und das Kompositum delegiert weiter.

Verwendung von Mustern

Möglichst schon bekannte Muster verwenden.

Vorteile:

- Vermeidung von Logikfehlern (denn Muster sind vielfach erprobt und haben sich bewährt)
- spart Zeit (muss nicht von vorne entwickeln -> auf Erfahrung anderer stützen)

Die Herausforderung ist, Muster auf eigene Porbleme anzupassen. Jedoch falsche Sicht: nicht designen mit Vorsatz 'ich muss Pattern verwenden', sondern Pattern verwenden, wenn sinnvoll erscheint.

Nachteile:

- Erzeugung zusätzlicher Klassen

  => führt zu Unübersichtlichkeit

Literatur

Anja, Metzner. 2020. *Software Engineering Kompakt.* Carl Hanser Verlag.

Helmut, Balzert. 2011. *Lehrbuch Der Softwaretechnik: Entwurf, Implementierung, Installation Und Betrieb.* Springer-Verlag.