Informatik
Hauptcampus

HOCHSCHULE TRIER

Titel der Arbeit

Bearbeiter 1: Vorname Nachname
Bearbeiter 2: Vorname Nachname
Bearbeiter 3: Vorname Nachname

Gruppe: GruppenID

Verbesserung und Evolution von Architekturen

Ort, Abgabedatum

Kurzfassung

In der Kurzfassung soll in kurzer und prägnanter Weise der wesentliche Inhalt der Arbeit beschrieben werden. Dazu zählen vor allem eine kurze Aufgabenbeschreibung, der Lösungsansatz sowie die wesentlichen Ergebnisse der Arbeit. Ein häufiger Fehler für die Kurzfassung ist, dass lediglich die Aufgabenbeschreibung (d.h. das Problem) in Kurzform vorgelegt wird. Die Kurzfassung soll aber die gesamte Arbeit widerspiegeln. Deshalb sind vor allem die erzielten Ergebnisse darzustellen. Die Kurzfassung soll etwa eine halbe bis ganze DIN-A4-Seite umfassen.

Hinweis: Schreiben Sie die Kurzfassung am Ende der Arbeit, denn eventuell ist Ihnen beim Schreiben erst vollends klar geworden, was das Wesentliche der Arbeit ist bzw. welche Schwerpunkte Sie bei der Arbeit gesetzt haben. Andernfalls laufen Sie Gefahr, dass die Kurzfassung nicht zum Rest der Arbeit passt.

The same in english.

Inhaltsverzeichnis

1

Architectural Tactics

1.1 Architektur-Ebene: Die SOLID-Prinzipien

Nicht unbedingt OOP. Ziel: SWA, die Modifikationen unterstützt. Prinzipien nicht neu, Anordnung zur Abkürzung durch Robert C. Martin.

1.1.1 Das Single Responsibility Princip (SRP)

„Ein Modul sollte für einen, unr nur einen, Akteur verantwortlich sein. “([Mar18], S.86)

1.1.2 Das Open-Closed Prinzip (OCP)

Bertrand Meyer
Veränderungen des Verhaltens können durch Codeergänzungen realisiert werden, keine Code-Modifikationen

1.1.3 Das Liskov'sche Substitutionsprinzip (LSP)

Barbara Liskov 1988
Elemente sind austauschbar

1.1.4 Das Interface-Segregation Prinzip (ISP)

Keine Abhängigkeiten von nicht genutzen Modulen

1.1.5 Das Dependency-Inversion Prinzip (DIP)

Code der übergeordnete Richtlinien implementiert sollte nicht von code abhängen, der untergeordnete Deatils implementiert.

1.2 Komponenten-Ebene: Prinzipien der Komponentenkohäsion

1.2.1 Das Reuse-Release-Equivalence-Prinzip (REP)

1.2.2 Das Common-Closure-Prinzip (CCP)

1.2.3 Das Common-Reuse Prinzip (CRP)

1.3 Komponenten-Ebene: Prinzipien der Komponentenkopplung

1.3.1 Das Acyclic-Dependencies-Prinzip (ADP)

1.3.2 Das Stable-Dependencies-Prinzip (SDP)

1.3.3 Das Stable-Abstractiond-Prinzip (SAP)

1.4 Weitere Design-Prinzipien auf Komponentenebene

1.4.1 Encapsulation (Kapselung)

Bei objektorientierter Programmierung Attribute von Klassen sowie Utility Functions private machen ([Mar09], S.136). So kann nicht von außen auf klasseninterne Implementierungsdetails zugegriffen werden. Wenn solche internen Details geändert werden, hat dies bei umgesetzter Kapselung keine Auswirkung auf andere Klassen.

1.4.2 Separation of Concerns (Trennung der Verantwortlichkeiten)

Jedes Systemelement hat eine klare Verantwortlichkeit. Änderungen an einer Systemoperation wirken sich somit nur auf dieses Systemelement aus. Sind hingegen viele Elemente für eine Operation verantwortlich, werden Änderungen an der Operation weite Teile des Systems betreffen.[NR12]
[MEM04]

1.4.3 Functional Cohesion (Funktionale Kohäsion)

Eine hohe Kohäsion bedeutet, dass die Methoden und Variablen einer Klasse voneinander abhängen und logisch zusammengehören. [Mar09], S.140. Demnach spricht eine hohe Kohäsion dafür, dass das System logisch sinnvoll in Einheiten unterteilt ist.
Funktionale Kohäsion liegt dann vor, wenn die Teile eines Moduls alle zur Lösung einer einzelnen, wohldefinierten Aufgabe beisteuern.
Dies führt zu vielen kleinen Klassen.

1.4.4 Single Point of Definition

Datentypen, Werte, Algorithmen, Konfigurationen Schemata etc. werden nur einmal definiert und implementiert ([NR12], S.551). D.h. wenn sie geändert werden müssen, werden sie nur einmal geändert.

1.5 Design-Patterns

Erlauben es, die Auswirkungen von Änderungen lokal zu begrenzen.

1.5.1 Creational Patterns

Abstract Factory (Abstrakte Fabrik)

[Mar18], S.109 Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.
Use the Abstract Factory when your code needs to work with various families of related products, but you don?t want it to depend on the concrete classes of those products?they might be unknown beforehand or you simply want to allow for future extensibility.
from https://refactoring.guru/design-patterns/abstract-factory

1.5.2 Structural Patterns

Facade (Fassade)

[Mar18], S.89 Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. https://refactoring.guru/design-patterns/facade
Reduziert Abhängigkeiten von Library, falls Library sich ändert

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.
Erlaubt es Extension Points zu verwenden

1.5.3 Behavioural Patterns

Mediator

Iterator

1.5.4 Weitere Patterns

Dependency Injection

[Mar09], S. 157
Wir fassen nochmal zusammen: DI bedeutet eigentlich nichts anderes, als Abhängigkeiten

zu konkreten Implementierungen aufzulösen indem man z.B. mit Interfaces arbeitet. Das sorgt dafür, dass die aufrufende Komponente oder Klasse die Abhängigkeiten an die aufgerufene Komponente oder Klasse zur Laufzeit übergibt.Bleibt noch die Frage, wer das eigentlich braucht. Das einfachste Beispiel ist der Fall von Applikationen, die auf unterschiedlichen Plattformen laufen sollen, die für bestimmte Systemaufrufe unterschiedliche APIs anbieten. Wer hier mit DI arbeitet, der kann plattformspezifisches Verhalten wegabstrahieren und so sehr viel gemeinsamen Quellcode nutzen. Plattformspezifika werden dann zur Laufzeit via DI eingefügt.
https://www.microsoft.com/de-de/techwiese/know-how/was-ist-eigentlich-dependency-injection.aspx

Extension Interface

1.6 Metamodel-basierte Architekturstile

1.7 Variation Points

Lokale Design-Lösungen, um bestimmte Änderungen an bestimmten Stellen im System zu unterstützen, wobei der Begriff ursprünglich aus der Produktlinien-Architektur stammt ([NR12], S.554).
Hier auch Kosten-Nutzen abwägen, ob die Variation Points wirklich gebraucht werden.

1.7.1 Elemente austauschbar machen

Implementierung und Interface getrennt halten, s.d. die Implementierung jederzeit ausgetauscht werden kann (Abhängigkeit nur vom Interface)([NR12], S.554).

1.7.2 Konfigurationen verwenden

Systemverhalten durch Parametrisierung steuern ([NR12], S.554). Dann können einfach die Parameter ausgetauscht werden.

1.7.3 Self-Describing Data und Generic Processing

Anstatt hard-coden mitgelieferte Informationen über die Daten nutzen, um diese generisch zu verarbeiten ([NR12], S.554).

1.7.4 Physische und Logische Verarbeitung getrennt halten

Einfacher wenn physiches Format sich ändert, z.B. CSV zu XML ([NR12], S.554)

1.7.5 Prozesse in Schritte unterteilen

Dann lassen sich die einzelnen Schritte leichter abändern ([NR12], S.554)
[DCLTQ15]

1.8 Extension Points

In Standard-Technologien bereits eingebaute Möglichkeiten, um Änderungen zu unterstützen ([NR12], S.554). Beispielsweise Custom Adapter schreiben

1.9 Sicherheit bei Änderungen

1.9.1 Konfigurationsmanagement

1.9.2 Automatisierte Build-Prozesse

1.9.3 Dependency analysis

1.9.4 Automatisierte Release Prozesse

1.9.5 Zurücksetzen

1.9.6 Environment Konfigurationsmanagement

1.9.7 Automatisiertes Testen

1.9.8 Continuos Integration

1.9.9 Entwicklungsumgebung sichern

Literaturverzeichnis

DCLTQ15. Di Cola, Simone, Kung-Kiu Lau, Cuong Tran und Chen Qian: *Defining Logical Architectures with Variation Points*. 10 2015.

Mar09. Martin, Robert C.: *Clean Code: a handbook of agile software craftsmanship*. Prentice Hall Pearson Education, 2009.

Mar18. Martin, Robert C.: *Clean Architecture: Das Praxis-Handbuch für professionelles Softwaredesign, 1. Auflage*. mitp Verlag, 2018.

MEM04. Mili, Hafedh, Amel Elkharraz und Hamid Mcheick: *Understanding separation of concerns*. 01 2004.

NR12. Nick Rozanski, Eoin Woods: *Software Systems Architecture, Second Edition*. Addison-Wesley, 2012.

A

Glossar

DisASTer	DisASTer (Distributed Algorithms Simulation Terrain), A platform for the Implementation of Distributed Algorithms
DSM	Distributed Shared Memory
AC	Linearisierbarkeit (atomic consistency)
SC	Sequentielle Konsistenz (sequential consistency)
WC	Schwache Konsistenz (weak consistency)
RC	Freigabekonsistenz (release consistency)