embarc Software Consulting GmbH
SIGS DATACOM FACHINFORMATIONEN FÜR IT-PROFESSIONALS

architektur SPICKER

Übersichten für die konzeptionelle Seite der Softwareentwicklung

MEHR WISSEN IN KOMPAKTER FORM:
Weitere Architektur-Spicker gibt es als kostenfreies PDF unter www.architektur-spicker.de

NR. 8

Nachhaltiges Software-Design

Dieser Spicker unterstützt Sie und Ihr Team bei der Auswahl und Umsetzung zeitgemäßer Design-Prinzipien und dem Entwurf einer nachhaltigen Software-Architektur.

IN DIESER AUSGABE

Worum geht's?

- Wie vermeiden Sie steigende Aufwände in der Wartungstätigkeit durch fortschreitende Erosion der Codestrukturen?
- „If you cant measure it, you can´t manage it." (P. Drucker) – Wie lässt sich Entwufsqualität messen?
- Die SOLID Prinzipien stammen aus dem Jahre 2000. Sind sie noch zeitgemäß oder müssen wir inzwischen weiterdenken?

Warum gezieltes Software-Design?

Durch Zerlegung in einzelne Bausteine bleibt Ihre Software langfristig änderbar. Die Fehleranfälligkeit bei Releases sinkt und Teams bleiben effizient in der Weiterentwicklung. Die einzelnen Bausteine:

... sind möglichst ohne unerwünschte Seiteneffekte zu ändern.

... können unabhängig voneinander dokumentiert und verstanden werden.

... sind an ihren ein- und ausgehenden Schnittstellen isoliert testbar.

... sind einzeln bei Bedarf austauschbar.

Software-Design einst und jetzt

SOLID (Robert C. Martin „Design Principles and Design Patterns", 2000) umfasst 5 Design-Prinzipien mit dem Fokus auf objektorientierte Programmierung. Im Zeitalter von Microservices sollten Sie weiterdenken, denn:

- SOLID ist nicht vollständig. Es fehlt z.B. das Prinzip des Information-Hiding - eine der wichtigsten Handlungsmaximen beim Modulentwurf.
- SOLID setzt den Fokus auf die damals vorherrschende objektorientierte Programmierung (OOP).
- Das O (für Open-Closed) und das D (für Dependency-Inversion) in SOLID interpretieren wir heute anders als damals.
- L (Liksovsches-Substitutionsprinzip) und I (Interface-Segregation) aus SOLID adressieren effizientes Schnittstellendesign doppelt, aber trotzdem nicht vollständig oder gut priorisiert.

Die 5 Schritte zum Erfolg beim Entwurf Ihrer Bausteine:

Das hier vorgestellte 5C-Modell fasst vorhandene Designprinzipien zeitgemäß zusammen, und stellt eine Kategorisierung von Handlungsmaximen dar.

Sie können es aber auch als eine Art Vorgehensmodell sehen, bei der Sie Ihre Bausteinzerlegung in dieser Reihenfolge festlegen.

- **Cut - Richtig Schneiden**
- **Conceal – Verbergen**
- **Contract – Schnittstelle festlegen**
- **Connect – Verbinden**
- **Construct – Aufbauen**

Untermauert wird jeder der 5 Schritte durch eine Zieldefinition, damit verwandte Prinzipien, Vorschläge zur Umsetzung (wie Pattern) und Kennzahlen zur Messung des Erfolgs.

Cut – Richtig Schneiden

Je unabhängiger ein Baustein ist, desto einfacher wird er zu handhaben sein. Schneide Bausteine so, dass sie sich möglichst gut voneinander abgrenzen.

Ziele dieses Schrittes

- Ein Baustein erfüllt nur eine konkrete Aufgabe, womit es nur einen Grund gibt, diesen zu ändern (Separation-of-Concerns/Single-Responsibility)
- Der innere Zusammenhalt (Kohäsion) ist möglichst groß, was zu einer geringeren Kopplung (siehe Schritt **Connect**) führt.

Verwandte Prinzipien

- Single-Level-of-Abstraction: Unterschiedliche Abstraktionsebenen der Verarbeitung sollen in verschiedenen (Sub-)Bausteinen gekapselt sein.

Umsetzung - klassisch

- Mit horizontalen Strukturen (Layern) betonen Sie die technischen Aspekte des Codes, mit Vertikalen (Slices) dagegen die Fachlichen.

Umsetzung – heute

- Im Strategic Design aus dem Domänengetriebenen Entwurf (DDD) wird explizit die Aufteilung der Business-domäne in ihre einzelnen Teilbereiche (Subdomänen) forciert.
- Event-Storming: Die idealen Grenzen der Subdomänen werden in diesen Workshops gemeinsam mit den Domänenexperten gefunden.

Messung des Erfolges

- LCOM4: Eine Zahl > 1 zeigt an, dass eine Klasse mehr als eine Aufgabe hat.
- Relational-Cohesion: Ein niedriger Wert ist ein Anzeichen für eine schwache innere Kohäsion.
- Zyklomatische-Komplexität: Ein zu großer Wert bedeutet, dass ein Baustein zu komplex ist und aufgeteilt werden soll.

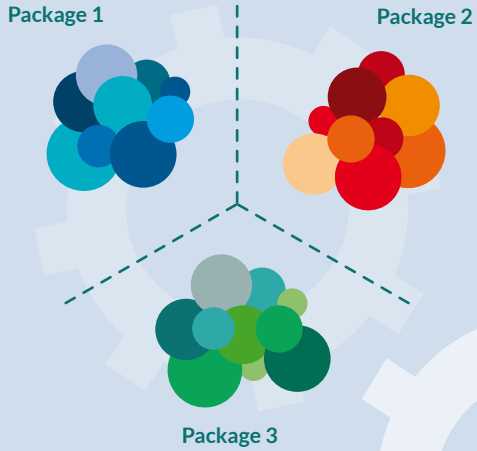

Abb. 1: Trennen, was nicht zusammengehört

Conceal - Verbergen

Verbirg so viel der internen Struktur eines Bausteins und der Art der Umsetzung vor der Außenwelt wie möglich.

Ziele dieses Schrittes

- Das Innenleben eines Bausteins kann ohne Abstimmungsaufwände, und auch ohne Gefahr von unerwünschten Seiteneffekten angepasst werden.

Verwandte Prinzipien

- Information-Hiding-Principle: Je größer die Wahrscheinlichkeit ist, dass sich etwas ändert, desto eher sollte es verborgen sein.
- Law-of-Demeter/Principle-of-Least-Knowledge: Das Zusammenspiel eines Bausteins A mit anderen Bausteinen soll für jemanden der Baustein A benützt nicht ersichtlich sein.

Umsetzung - klassisch

- Bord-Mittel einiger Programmiersprachen: Klassen, package-protection und JigSaw Module in Java. Klassen und Assemblies in C#. ES6 Module in JavaScript.
- Design-Pattern: Facade, Iterator

Umsetzung – heute

- Microservices: Durch Abtrennung eines Service und Kommunikation über Netzwerkschnittstellen werden Interna vor den Consumern verborgen.

Messung des Erfolges

- Visibility-Metriken: Niedrige Werte für Relative Visibility, Average Relative Visibility und Global Relative Visibility sind anzustreben

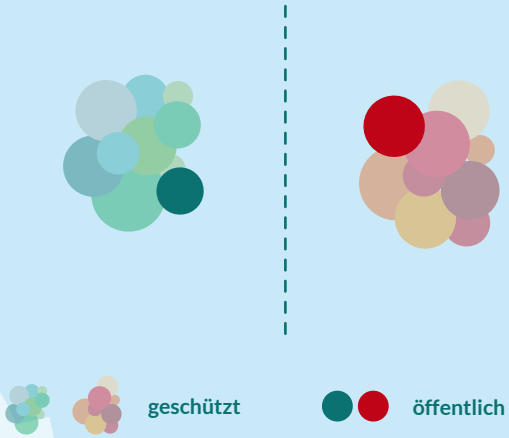

Abb. 2: So viel verbergen wie möglich

Contract – Schnittstelle festlegen

Entwerfe Schnittstellen so, dass eine möglichst reibungslose Interaktion zwischen dem Baustein und seinen Consumern möglich ist.

Ziele dieses Schrittes

- Die Schnittstelle ist für den Consumer einfach zu verstehen und anzuwenden.
- Eine missbräuchliche Verwendung der Schnittstelle ist kaum möglich.
- Die Schnittstelle ist ausreichend konkret spezifiziert und dokumentiert.
- Die Schnittstelle ist schmal und dem jeweiligen Anwendungsfall angemessen.

Verwandte Prinzipien

- Open-Closed: Schnittstellen dienen zur Erweiterung der Funktionalität von Bausteinen, ohne dass bestehender Code angepasst werden muss.
- Interface-Segregation: Verschiedene Verwendungs-szenarien werden in unterschiedlichen Schnittstellen gekapselt.
- Liskovsches-Substitutionsprinzip: Eine Unterklasse darf den impliziten Kontrakt einer Oberklasse nicht ändern.
- Postel´s-Law/Robustheitsprinzip: Sei konservativ bei dem, was du sendest, aber liberal bei dem, was du akzeptierst.

Umsetzung – klassisch

- Design-by-Contract: Explizites Prüfen von ein- und ausgehenden Parametern macht den impliziten Teil einer Schnittstelle expliziter.
- Design-Pattern: Decorator, Observer, Strategy, Bridge

Umsetzung – heute

- Durch den REST-Interaktionsstil werden komplexe Schnittstellen automatisch in einzelne Entitäten aufgeteilt, währen die Art der Interaktion standardisiert ist.
- Consumer-Driven-Contracts: Durch Testen der Erwartungen der Consumer können Änderungen am Provider ohne Abstimmungsaufwände oder Integrationstests laufend released werden.

Messung des Erfolges

- Depth-of-Inheritance: Eine tiefe Vererbungskette von Klassen zeigt eine übertriebene Anwendung von Vererbung an. Vererbung in der OOP stellt eine eher intransparente Form einer Schnittstelle dar.

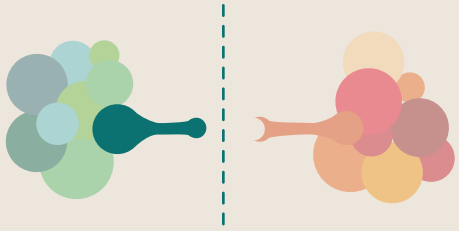

Abb. 3: Funktionalität über Schnittstellen anbieten und einfordern

Connect – Verbinden

Durch Verwendung einer Schnittstelle eines anderen Bausteins kommt es immer zu Abhängigkeiten. Plane explizit zwischen welchen Bausteinen es welche Art von Abhängigkeit geben soll.

Ziele dieses Schrittes

- Es herrscht lose Kopplung und es gibt wenig einschränkende Auswirkungen der Verbindungen.
- Risiken und Probleme pflanzen sich nicht durch kaskadierende Abhängigkeiten fort.
- Es herrscht Ausgewogenheit, da es keine zentralen Bausteine mit sehr vielen Abhängigkeiten gibt.

Verwandte Prinzipien

- Dependency-Inversion: Jede Form der Abhängigkeitsumkehr
- Inversion-of-Control: Konkrete Form der Dependency-Inversion wo die Kontrolle über das Zusammenspiel der Bausteine an eine übergeordnete Instanz (meist ein Framework) delegiert wird.
- Dependency-Injection: Konkrete Form der Inversion-of-Control. Delegieren der Erzeugung von abhängigen Instanzen an ein allgemeines Framework.

Umsetzung – klassisch

- Prinzip der azyklischen Abhängigkeiten
- Design-Pattern: Adapter, Proxy, Mediator

Umsetzung – heute

- Message-Broker: Gepufferte Kommunikation zwischen Services reduziert zeitliche Abhängigkeit.
- Timeout, Circuit-Breaker und Bulkhead-Pattern: Die kaskadierende Ausbreitung von Fehlersituationen im System über Abhängigkeiten wird verhindert.
- DDD Bounded Context: Abgrenzung der Modelle und Integration durch Konzepte wie den „Anti-Corruption-Layer.
- BASE Consistency und Saga-Pattern: Durch Verzicht auf strenge ACID Konsistenz ist eine technisch losere Kopplung zwischen Services möglich.

Messung des Erfolges

- Metriken von John Lakos: RACD und NCCD summieren die Kopplungen innerhalb des Systems.
- Stability aus den Software-Package Metriken: Viele Abhängigkeiten sorgen für Instabilität eines Bausteins bei Änderungen.

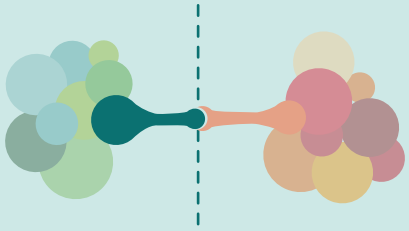

Abb. 4: Anbieter und Nutzer verbinden

Construct - Aufbauen

Eine Bausteinstruktur kann auf einer Ebene selbst wieder unübersichtlich werden. Baue Systeme höherer Komplexität durch Zusammenfassen von Bausteinen einer Ebene zu einem neuen Baustein einer nächsthöheren Ebene. Dabei sind weiterhin dieselben Handlungsmaximen anzuwenden.

Ziele dieses Schrittes

- Komplexere Systemlandschaften werden dadurch wartbarer.
- Eine aus dem Ufer geratene Komplexität kann den Unternehmenserfolg nicht gefährden.
- Verringerung von Abstimmungsaufwänden und Bürokratie.

Verwandte Prinzipien

- Teile und herrsche

Umsetzung - klassisch

- Bord-Mittel der Programmiersprachen wie Module, Packages und Klassen in Java.
- Das C4 Modell von Simon Brown stellt eine Möglichkeit dar, hierarchische Architekturen umzusetzen und zu beschreiben.
- Design-Pattern: Whole-Part

Umsetzung - heute

- Self-Contained-Systems sind grobgranularer als Microservices und können Microservice-Systeme mit modularen Monolithen kombinieren.

Messung des Erfolges

- Automatisierte Prüfung der Bausteinstruktur mit Hilfe von Tools wie ArchUnit, Sonargraph oder Teamscale

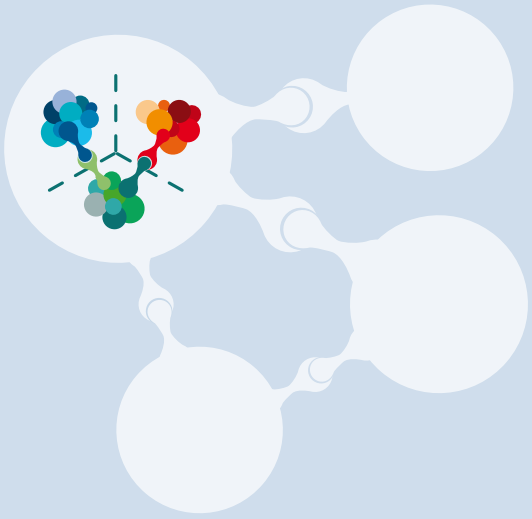

Abb. 5: Die fraktale Natur eines Designs

Weitere Informationen

Bücher (Auswahl)

- Erich Gamma et al.: „Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software“, mitp Professional 2014
- Herbert Dowalil: „Grundlagen des modularen Softwareentwurfs“, Carl-Hanser Verlag 2018
- Eric Evans: „Domain-Driven Design: Tackling Complexity in the Heart of Software“, Addison Wesley 2013
- Chris Richardson: „Microservice Patterns: With examples in Java“, Manning 2018
- Stefan Tilkov et al.: „Rest und HTTP Entwicklung und Integration nach dem Architekturstil des Web“, dpunkt.verlag GmbH 2015

Der Autor dieses Spickers

Herbert Dowalil ist Softwareentwickler und -architekt bei embarc in Wien.
Kontakt: herbert.dowalil@embarc.at | Twitter: @hdowalil

Nützliche Links

- Werkzeuge zur Messung des Erfolges: https://embarc.de/metriken-tools-blog
- Design Principles and Design Patterns, Robert C. Martin, 2000: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf
- Visibility Metrics: https://dzone.com/articles/visibility-metrics-and-the-importance-of-hiding-th
- Über die Vermessung von Software: https://www.embarc.de/ueber-die- vermessung-von-software
- The C4 model for software architecture: https://c4model.com
- Self-Contained Systems: https://scs-architecture.org/
- Distributed Tracing mit Sleuth und Zipkin: https://spring.io/blog/2016/02/15/distributed-tracing-with-spring-cloud-sleuth-and-spring-cloud-zipkin

Wir freuen uns auf Ihr Feedback: spicker@embarc.de

https://architektur-spicker.de

embarc Software Consulting GmbH
https://www.embarc.de
info@embarc.de

SIGS DATACOM
FACHINFORMATIONEN FÜR IT-PROFESSIONALS
https://www.sigs-datacom.de
info@sigs-datacom.de