AES

Domain-Driven Design

* ~~Entity~~
* ~~Lokale und globale Identität~~
* ~~Assoziationen zwischen Aggregates~~
* ~~Anti-Corruption Layer~~
* ~~Bounded Context~~
* ~~Context Map~~
* ~~Lebenszyklus eines Domain Objekts~~

Software Architektur und Design Prinzipien

* ~~4+1 Sichten einer Architektur~~
* ~~Prinzip Information Hiding~~
* ~~Was spricht für die Zerlegung einer Anwendung in Komponenten~~
* ~~Welches Merkmal einer Komponente gibt Aufschluss darüber wie gut eine Komponente wiederverwendet werden kann~~
* ~~Inversion Control und Dependency Injection~~
* ~~Service Locator~~
* ~~Warum kann man mit Kapselung die Einhaltung von Invarianten leichter sicherstellen~~
* ~~Liskovsche Substitutionsprinzip~~
* ~~Interface-Segregation Prinzip~~
* ~~Konfiguration und Konfigurationsmanager~~
* ~~Konzept Design by Contract~~ (irrelevant)
* ~~Warum Verwendung von globalen Variablen problematisch im Hinblick auf Invarianten ist~~
* ~~Komponentenarchitektur~~
* ~~Dependency Inversion~~

Struktursicht

* ~~A und T Sfotware immer trennen~~
* ~~R Software~~
* ~~Sichtbarkeitsregeln~~
* ~~Unreine Softwarekategorie~~
* ~~Möglichkeiten wie Kopplung bei Schnittstellenentwurf beeinflusst wird~~
* ~~Kommunikation mit neutralen Schnittstellen~~

Prozesssicht und Skalierbarkeit

* ~~Session-Context Manager Pattern~~ (irrelevant)
* ~~Skalierungsstrategie Scale Out~~
* ~~Asynchrone Kommunikation~~
* ~~Was passiert wenn Queue voll ist~~
* ~~Worker pool pattern~~

Physische Sicht

* ~~Active Cluster skizzieren und erklären~~
* ~~Fail-Over Cluster~~
* ~~Application-Cluster~~

Design Pattern, Enterprise Pattern

* ~~Welche Grundidee wird durch den Einsatz von Erzeugermustern gelöst~~
* ~~Interpreter Pattern~~
* ~~Strategy Pattern~~
* ~~Logical Unit of Work~~
* ~~Objectpool~~
* ~~Visito Pattern~~
* ~~Welches Pattern findet man, wenn man mit Collections arbeitet~~ (irrelevant)
* ~~Prinzip der Abstraktion~~
* ~~Proxy Pattern~~

Verhaltensmuster

Erzeugermuster

Logische Sicht

* ~~Aggregates~~
* ~~Root Entry~~
* ~~Intension Revealing Interfaces~~

5. zerkegt monolithen in einzelne services, services können in eigenständigen technologien von eigenständigen teams entwickelnwerden, mehr unabhängigkeit, bessere release zyklen, bessere einzelne komponenten zu skalieren

B wie machen wir einen security kontext wenn

1.b)

- ein System muss immer mit Altsystemen bzw. existierenden Systemen integriert werden

- Modelle von Altsystemen oftmals widersprüchlich und labil

Lösung:

* Isolation Layer definieren welche dem Client externe Funktionalität zur Verfügung stellt
* Layer kommuniziert mit externen Systemen durch existierende Schnittstellen
* Intern ist das Layer ein Übersetzer in beide Richtungen
* Applikation bleibt unabhängig von dem angebundenen System
* Kann später durch neue Implementierungen ersetzt werden

1.c)

- Klassen und Operationen benennen um Zweck und Wirkung zu beschreiben, ohne Bezug auf Realisierung zu nehmen

- dadurch werden Internas von Nutzern verborgen

- negatives Beispiel: doSomethingWithKapitel()

- positives Beispiel: ExportChapterAsPDF()

2.a)

- stimmt nicht

- kann auch in C umgesetzt werden

- Bibliothek statt Komponenten

2.b)

- die Komponenten wissen nichts über ihre Umgebung

- die Umgebung steuert die Komponenten und deren Beziehung zueinander

- Service Locator und Dependency Injection

2.c)

- Dieses Strukturierungsprinzip benötigt etwas mehr overhead da schnittstellen und Komponenten defineirt werden müssen

- Dieser Overhead ist gering wenn es sich um ein größeres Projekt handelt.

- Dieser Ansatz garantiert höhere Flexibilität und bessere Erweiterbarkeit

- Beugt Fehler durch die Reduktion von Kopplung vor

2.d)

- Software-Einheiten sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein  
- Eine Erweiterung in diesem Sinne verändert das vorhandene Verhalten der Einheit nicht, vielmehr erweitert es die Einheit um zusätzliche Funktionen oder Daten   
- Eine Modifikation hingegen, würde das bisherige Verhalten der Einheit ändern

2.e)

- Alle beweisbaren Eigenschaften der Oberklasse müssen auch für

- Alle beweisbaren Eigenschaften der Oberklasse müssen auch für alle Unterklassen gelten.  
- Wird gegen das Prinzip verstoßen ist die polymorphe Verwendung problematisch weil sich Unterklassen anders verhalten als die Oberklassen. Dadurch kann es zu instabilen Programmsystemen kommen

- Notwendig: Durch die Einhaltung des Prinzips kann sich ein Client darauf verlassen dass jedes abgeleitete Objekt genauso benutzt werden kann wie die Oberklasse selber