Recap

Architekturanalyse bis Testing

Software Architecture

- Gesamtheit der wichtigen Entwurfs-Entscheidungen
- Definiert tragende Elemente der Architektur
- Wir haben immer eine Architektur, ob bewusst oder unterbewuesst
- Focus auf zukünftige Änderungen (unterstützt refaktoring)
- Top level view: grosse Ganze

Architekturanalyse

- Wie kommen wir zur richtigen Architektur?
- Grundlage: Anforderungen (insbesondere nicht-funktionale) und Systemkontext mit Schnittstellen
- Twin Peaks Modell: Qualität und Stabilität der Anforderungen selbst überprüfen

GRASP Prinziples

Pattern/Prinzip	Beschreibung
Information Expert	Ein allgemeines Prinzip des Objektentwurfs und der Zuweisung von Verantwortlichkeiten; weisen Sie Verantwortlichkeiten dem *Information Expert* zu – der Klasse, die über die zu deren Erfüllung erforderlichen Informationen verfügt.
Creator	Wer erstellt? (Anmerkung: *Factory* ist eine gebräuchliche alternative Lösung.). Übertragen Sie Klasse B die Verantwortung, eine Instanz von Klasse A zu erstellen, wenn eine der folgenden Aussagen wahr ist: 1. B enthält A. – 2. B aggregiert A. – 3. B enthält Initialisierungsdaten für A. – 4. B zeichnet A auf. – 5. B ist mit A bei der Verwendung eng verbunden.
Controller	Welches erste Objekt jenseits der UI-Schicht empfängt und koordiniert (»kontrolliert«) eine Systemoperation? Weisen Sie die Verantwortlichkeit einem Objekt zu, auf das eine der folgenden Aussagen zutrifft: Repräsentiert das Gesamtsystem; ist ein »Root«-Objekt; ist ein Gerät, in dem die Software läuft; ist ein größeres Subsystem; (dies sind alles Varianten von *Facade Controller*). Repräsentiert ein Use-Case-Szenario, in dem die Systemoperation abläuft (ein *Use Case* oder *Session Controller*).
Low Coupling (evaluativ)	Wie kann der Einfluss von Änderungen verringert werden? Weisen Sie Verantwortlichkeiten so zu, dass die (nicht erforderliche) Kopplung gering bleibt. Verwenden Sie dieses Prinzip, um Alternativen auszuprobieren.
High Cohesion (evaluativ)	Wie kann man Objekte fokussiert, verstehbar und handhabbar halten und nebenbei *Low Coupling* unterstützen? Weisen Sie Verantwortlichkeiten so zu, dass der Zusammenhang (die Kohäsion) hoch bleibt. Verwenden Sie dieses Prinzip, um Alternativen auszuprobieren.
Polymorphism	Wer ist verantwortlich, wenn das Verhalten je nach Typ variiert? Wenn verwandte Alternativen oder Verhaltensweisen je nach Typ (Klasse) variieren, weisen Sie Verantwortlichkeit für das Verhalten mit polymorphen Operationen den Typen zu, bei denen das Verhalten variiert.
Pure Fabrication	Wer ist verantwortlich, wenn Sie verzweifelt sind und die Prinzipien der *High Cohesion* und des *Low Coupling* nicht verletzen wollen? Weisen Sie einen hochkohäsiven Satz von Verantwortlichkeiten einer künstlichen oder geeigneten »Verhaltensklasse« zu, die kein Konzept des Problembereichs repräsentiert, sondern die willkürlich geschaffen wird, um eine hohe Kohäsion, eine geringe Kopplung und die Wiederverwendbarkeit zu unterstützen.
Indirection	Wie kann man Verantwortlichkeiten zuweisen, um eine direkte Kopplung zu vermeiden? Weisen Sie die Verantwortlichkeit einem zwischengeschalteten Objekt zu, das zwischen anderen Komponenten oder Diensten vermittelt, sodass diese nicht direkt gekoppelt sind.
Protected Variations	Wie weist man Verantwortlichkeiten Objekten, Subsystemen und Systemen so zu, dass Variationen oder Instabilitäten in diesen Elementen keinen unerwünschten Einfluss auf andere Elemente haben? Identifizieren Sie die Stellen, an denen Variationen oder Instabilitäten zu erwarten sind; weisen Sie Verantwortlichkeiten zu, um diese Stellen durch eine stabile »Schnittstelle« einzuschließen.

Design Modellierung

- Statische Modellierung
  - z.B. UML-Klassendiagramm
  - Pakete, Klassennamen, Attributen, Methodensignaturen (ohne Methodenkörper)

- Dynamische Modellierung
  - z.B. UML-Interaktionsdiagramm
  - Entwurf der Logik, Verhalten des Codes und Methodenkörper

Implementation

- Von Design to Code:
  - Design-Klassendiagramme → Klassennamen, Interfacenamen, Attribute, Methodensignatur, Superklassen (Ableitungen)
  - Interaktionsdiagramme → Methodenimplementation, Methodensignatur

Refactoring

- Strukturierte, disziplinierte Methode, vorhandenen Code umzuschreiben
- Externes Verhalten bleibt gleich!
- Viele kleine Schritte (Codeänderungen)
- Interne Struktur wird verbessert

Testing

- Grundlegende Testarten:
  - Funktionaler Test (Black-Box Verfahren)
  - Nicht funktionaler Test (Lasttest etc.)
  - Strukturbezogener Test (White-Box Verfahren)
  - Änderungsbezogener Test (Regressionstest etc.
- Weitere Teststufen/-arten:
  - Integrationstest
  - Systemtest
  - Abnahmetest
  - Regressionstes

Design Patterns

- Adaptor
- Factory
- Singleton
- Dependency Injection
- Proxy
- Chain of Responsibility
- Decorator
- Observer
- Strategy
- Composite
- State
- Visitor
- Facade