0.1 Objektorientierung

Grundidee: Reale Welt besteht aus Objekten, die untereinander in Beziehungen stehen.

- Klasse:
  - Daten(Attribute)
  - Funktionalität(Operationen, Methoden)
- Objekte:
  - In der Lage Nachrichten (= Methodenaufrufe) zu empfangen
  - Daten verarbeiten
  - Nachrichten senden
  - können einmal erstellt werden
  - in verschiedene Kontexten wiederverwendet werden

Objektorientierte Analyse(OOA): Objekte-Konzepte-in Domäne zu finden und zu beschreiben.

Objektorientierten Design (OOD): Geeignete Softwareobjekte und ihr Kollaboration zu definieren um Anforderungen zu erfüllen.

Domänenschicht: Klassen abgeleitet aus dem Domänenmodell (Low-Representational-Gap)

0.1.1 Use Cases und System-Sequenzdiagramm

Basis für das Design:

1. Szenarien
2. Systemoperationen
3. Domänenmodell

Was programmiert werden muss:

1. Systemoperationen bzw. deren Antworten

Use-Case-Realisierung: Wie ein bestimmter Use Case innerhalb Design mit kollaborierenden Objekten realisiert wird.

Systemoperationen: Jedes Szeanario schrittweise entworfen und implementiert

UML-Diagramme: Gemeinsame Sprache um Use-Case-Realisierungen zu veranschaulichen und zu diskutieren.

0.1.2 Klassen entwerfen:

Zwei arten von Design-Modellen (ergänzen sich und werden parallel erstellt):

- Statische Modelle:
  - UML-Klassendiagramm- Unterstützen Entwurf Paketen, Klassennamen, Atrributen und Methodensignaturen (ohne Methodenkörperd)
- Dynamische Modelle:
  - UML-Interaktionsdiagramme Untersützen Entwurf der Logik, des Verhaltens des Codes und der Methodenkörper.

0.2 UML-Diagramme für das Design

0.2.1 Grundelemente der UML

- Primitiver Datentyp
- Literal
- Schlüsselwort, Stereotyp
- Randbedingung (constraint)
- Kommentar
- Diagrammrahmen (optional)

0.2.2 UML-Klassendiagramm

- Statische struktur
- Konzeptuell: Problemdomäne (Domänenmodell)
- Design: Lösungsdomäne (DCD)

Notationselemente

- Klasse
- Attribut
- Operation
- Sichtbarkeit von Attributen und Operationen
- Assoziationsname, Rollen an den Assoziationsenden
- Multiplizität (Objekte der betreffenden Klasse)
- Navigierbarkeit in Assoziationen
- Datentypen und Enumerationen
- Generalisierung / Spezialisierung
- Abstrakte Klassen
- Komposition
- Aggregation
- Interface
- Interface - Realisierung (Menge von öffentlichen Operationen, Merkmalen und Verpflichtungen, die durch eine Klasse, die die Schnittstelle implementiert, zwingend zur Verfügung gestellt werden müssen.
- Assoziationsklasse (Da wenn ** Beziehung existiert)
- Aktive Klasse (Instanz wird in einem separaten Thread ausgeführt)

0.2.3 UML-Interaktionsdiagramme

Spezifiziert, auf welche Weise Nachrichten und Daten zwischen Interaktionspartnern ausgetauscht werden.

2 Arten:

1. Sequenzdiagramm
2. Kommunikationsdiagramm

Anwendung: Kollaborationen bzw. Informationsaustausch zwischen Objekten zu modellieren.

- Wer tauscht mit Wem Information aus?
- in welcher Reihenfolge
- Zeitlicher Ablauf
- Schachtelung und Flussteuerung (Bedingung, Schleifen, Verzweigungen) möglich.

Kann in mehreren Perspektiven verwendet werden:

- Analyse
  - mit SSD Input-/Output-Erignisse (Systemoperationen mit Rückgabeantworten) für ein Szenario eines Use Cases modelliert
- Design
  - mit SSD Interaktion zwischen Objekten zur Realisierung eines konkreten Use-Case Szenarios zu modellieren

Notationselemente:

- Lebenslinie
- Aktionssequenz
- Synchrone Nachricht
- Antwortnachricht
- Gefundene, verlorene Nachricht
- kombiniertes Fragment
- Erzeugnis-, Löschereignis
- Selbstaufruf
- Interaktionsreferenz
- Lebenslinie mit aktiver Klasse
- Asynchrone Nachricht

0.2.4 UML-Kommunikationsdiagramm

Beantwortet Frage:
Wer kommuniziert mit werm? Wer arbeitet im System zusammen? Darstellung Informationsaustausch zwischen Kommunikationspartnern. Überblick im Vordergrund.

Notationselemente:

- Lebenslinie (Box)
- Synchrone Nachricht (=Aufruf einer Operation)
- Antwortnachricht (=Rückgabewert)
- Bedingte Nachrichten "[]"
- Iteration "*"
- Nummerierung der Nachrichten (Festlegung Zeitliche Abfolge)

- Systemoperation (msg1, ohne Nummer)
- Nummern gleicher Hierarchie(1,2,3...)
- Nummer unterer Hierarchie:
  * Werden innerhalb der Operation darüber ausgeführt (verschachtelt)
  * Operation 1.1:msg3 wird innerhalb von Operation 1:msg2 aufgerufen

0.2.5 UML-Zustandsdiagramm

Beantwortet zentrale Frage:
Weleche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, ...bei welchen Ereignissen annehmen?
Präzise Abbildung eines Zustandmodells (endlicher Automat):

- Zuständen
- Ereignissen
- Nebenläufigkeiten
- Bedingungen
- Ein- und Austrittsaktionen

Verwendung: Modellierung von Echtzeitsystemen, Steuerungen und Protokollen

Notationselemente:

- Start-, Endzustand
- einfacher Zustand
- Zusammengesetzter bzw. geschachtelter Zustand
- Transition
- Orthogonaler Zustand
- Parallelisierungsknoten
- Synchronisationsknoten
- Einstiepunkt
- Ausstigespunkt
- Unterzustandautomat
- Zusammengesetzter Zustand
- Flache und tiefe Historie

0.2.6 UML-Aktivitätsdiagramm

Beantwortet zentrale Frage:
Wie läuft ein bestimmter Prozess oder ein Algortithmus ab?
Detaillierte Visualisierung von Abläufen:

- Bedingungen
- Schleifen
- Verweigungen

Parallelisierung und Synchronisation von Aktionen möglich.

Notationselemente:

- Aktivität
- Aktionsknoten (Aktion)
- Objektknoten (Objekt)
- Entscheidungs- und Vereinigungsknoten
- Initialknoten
- Aktivitätsendknoten
- Partition / Swimlane
- Parallelisierungsknoten
- Synchronisationsknoten
- SendSignal-Aktion
- Ereignis- bzw. Zeitereignisannahmeaktion
- CallBehavior-Aktion

0.3 Verantwortlichkeiten und Responsibility-Driven-Design

: Methode über Entwurf Softwareklassen nachzudenken: **Verantwortlichkeiten, Rollen, Kollaborationsbeziehungen** = **RDD / Responsibility Driven Design**

Softwareobjekte haben Verantwortlichkeiten und arbeiten mit anderen Objekten zusammen. Verantwortlichkeiten werden durch Attribute und Methoden implementiert.
Kann auf jeder Ebene des Designs angwendet werden:

- Klasse
- Komponente
- Schicht

2 Ausprägungen von Verantwortlichkeiten:

Doing-Verantwortlichkeiten / Algorithmen, Code:

- Selbst etwas tun
- Aktionen anderer Objekte anstossen
- Aktivitäten anderer Objekte kontrollieren und steuern

Knowing-Verantwortlichkeit / Daten, Attribute:

- Private eingekapselte Daten
- Dinge kennen, die es ableiten oder berechnen kann
- Daten/Objekte zur Verfügung stellen, die aus bekannten Daten/Objekten abgeleitet oder berechnet werden können

0.3.1 GRASP- 9 Patterns

1. **Information Expert**
   - Derjenige, der was weiss
2. **Creator**
   - Zuständig erstellen Objekte(new())im Framework ABER delegiert als Dependency Injection
3. **Controller /Domain Controller = Service**
4. **Low Coupling**
   - Weniger Abhängigkeiten
5. **High Cohesion**
   - Abhängigkeiten zusammentun
6. **Polymorphism**
   - Unterschiedliche Objekte können gleiche Nachricht empfangen und interpretieren
7. **Pure Fabrication**
8. **Indirection**
9. **Protected Variations**

0.3.2 Information Expert

Lösung:
Verantwortlichkeit einer Klasse zuweisen, die über die erforderlichen Informationen verfügt, um sie zu erfüllen. Partielle Verantwortlichkeiten sind auch möglich.
Alternativen:
Low Coupling, High Cohesion (Künstliche Klasse)

0.3.3 Creator

Denn Nachteil von new..() = Kopplung
Lösung:
Klasse A zuständig eine neue Instanz einer **Klasse B** zu erzeugen, wenn eine oder mehrerere der folgenden Aussagen wahr ist(je mehr desto besser):

- A eine Aggregation oder ein Kompositum von B
- A registriert oder erfasst B-Objekte
- A arbeitet eng mit B-Objekten zusammen oder hat enge Kopplung
- A verfügt über Initialisierungsdaten für B (A ist Experte bezüglich Erzeugung von B)

Wenn mehrere Optionen anwendbar sind, Klasse A vorziehen, die ein Aggregat oder ein Kompositum ist.
Alternative:
Factory Pattern, Dependency Injection(DI)

0.3.4 Controller

Service = 1.Schicht = Domain Controller
Welches erste Objekt jenseits der UI-Schicht empfängt und koordiniert(kontrolliert) eine Systemoperation?
Lösung:
Verantwortlichkeit der Klasse zuweisen, der eine der folgenden Bedingungen erfüllt:

- Variante 1: Fassaden Controller:
  - Repräsentiert Root-Objekt", System bzw. übergeordnetes System auf dem die Sowftware läuft.
- Variante 2: Use Case Controller:
  - Pro Use-Case-Szenario eine künstliche Klasse, in der die Systemoperationen abläuft.

Controller macht selber nur wenig und delegiert fast alles!

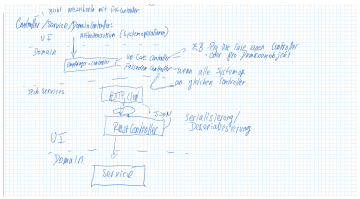

Abbildung 1: Controller.

0.3.5 Low Coupling

Ziel: geringe Abhängigkeit (besser für Refactoring)

- Kopplung = Mass für die Abhängigkeit von anderen Elementen (Klassen, Subsystem,Systeme)
- Hohe Kopplung= Element ist von vielen anderen Elementen abhängig
  - Nachteil: Refactoring mühsam, schwieriger zu verstehen, schwieriger wiederzuverwenden
- Niederige Kopplung = Element ist nur von wenigen anderen Elementen abhängig.

Lösung:
Verwantwortlichkeiten so zuweisen, dass Kopplung gering bleibt, Delegation Information Expert.

0.3.6 High Cohesion

Ziel:
Erreichen, dass Objekte fokussiert, verständlich handbar bleiben und nebenbei Low Coupling unterstützen

- Kohäsion = Mass für Verwandschaft und Fokussierung eines Elements
- Hohe Kohäsion = Element erledigt nur wenige Aufgaben, die eng miteinander verwandt sind
- Geringe Kohäsion = Element, das für viele unzusammenhängende Dinge verantwortlich ist

Nachteil geringe Kohäsion: Schwierig zu verstehen, schwierig wiederzuverwenden, brüchig und instabil, sind laufend von Änderungen betroffen
Lösung:
Verwantwortlichkeiten so zuweisen, dass Kohäsion hoch bleibt.

0.3.7 Polymorphismus

Ziel: Typabhängige Alternativen handhaben.

- Operation weist viele if-then-else bzw. grosse switch-case Anweisungen auf
- Bestimmtes Verhalten (z.B Einsatz eines externen Dienstes konfigurierbar machen.

Lösung:
Typabhängige Verhalten mit polymorphen Operationen der Klasse zuweisen, dessen Verhalten variiert.

- Grundlegende Idee OOP (Generalisierung / Spezialisierung)
- Überprüfen dass Beziehung ïs aïst zwischen Superklase und Subklassen
- Liskov-Substituions-Prinzip einhalten

0.3.8 Pure Fabrication

Problem: Wollen nicht gegen High Cohesion und Low Coupling oder andere Ziele verstossen, aber Lösungen nicht passen

- Viele Design-Klassen können direkt aus dem Fachbereich (Domänenmodell) abgeleitet werden und erfüllen das Low Representational Gap
- viele Situationen wo es Probleme gibt wenn Verantwortlichkeiten der Klasse in der Domänenschicht zugewiesen werden

Lösung:

- Hoch kohäsiven Satz von Verantwortlichkeiten einer künstlichen Hilfsklasse zuweisen
- Nur erstellt um höhere Kohäsion, geringe Kopplung oder eine bessere Wiederverwendbarkeit zu realisieren.

0.3.9 Indirection

Problem: Verantwortlichkeit zuweisen, so dass direkte Kopplung zwischen 2 oder mehrere Objekte vermeidet wird. Objekte Entkoppeln, für geringere Kopplung und Wiederverwendungspotentioal grösser.
Ziel: Zwischen Adapter vermittelt
Lösung:

- Verantwortlichkeit einem zwischengeschalteten Objekt zuweisen
- Vermittler schafft eine Indirektion zwischen den anderen Komponenten
- Alternativen = Protected Variations
- GoF Patterns wie Adapter, Bridge, Facade, Observer oder Mediator verwenden dieses Prinzip
- viele Indirections sind Pure Fabrications

0.3.10 Protected Variations

Problem: Objekte, Subsystem und Systeme entwerfen, so dass Veränderungen und Instabilitäten in diesen Elementen keinen Einfluss auf andere Elemente haben.
Ziel: Im Vorfeld Erweiterungen vorbereiten für Zukunft.

Lösung:

Punkte identifizieren, an denen Veränderungen und Instabilitäten zu erwarten sind. Verantwortlichkeiten so zuweisen, dass diese Punkte durch ein stabiles Interface eingekapselt werden.
Unterscheidung der Änderungspunkten:

- Variationspunkt:
  - Veränderung sind sicher(in Anforderung), zwingend PV Konzepte einbauen
  - Entwicklungspunkt:
    - Veränderungen nicht sicher, treffen mit hoher Wahrscheinlichkeit ein, sind nicht in Anforderungen enthalten.

Spekulative Anwendungen vermeiden, sonst unnötige Komplexität.