Advanced Software Engineering 2

        - Large Class
        - Shotgun Surgery
        - Switch Statements
        - Code Comments
        - Refactorings
            - Extract Method
            - Rename Method
            - Replace Temp with Query
            - Replace Conditional with Polymorphism
            - Replace ErrorCode with Exception
            - Replace Inheritance with Delegation
    - GUI Blooper
        - Was sind GUI-Blooper?
        - Durch Blooper verursachte Probleme
        - Gründe für Blooper
            - GUI Komponenten
                - Navigation
                - Text
            - Design und Layout
            - Interaktion
            - Management
            - Antwortverhalten
            - Responsiveness
            - Vermeidung von langem Antwortverhalten
                - Timely Feedback
                - Parallel Problem Solution
                - Queue Optimization
                - Dynamic Time Management
    - UI Entwicklung
        - Sinnvoller Designprozess
        - Definition von Usability
            - Wesentliche Elemente
            - Wesentliche Messgrößen
            - 5Es
            - User Centered Design Process
            - Analyse
                - Informationsquellen
                - Informationen zum Benutzer
                - Mentales Modell

        - Objekt-Aktions-Analyse
        - Regeln an ein mentales Modell
    - Lexikon
    - Szenarios
        - Gemeinsame Diskussionsgrundlage
        - Vorteile
    - Design
        - Umfang der Software
        - Anzahl Features versus Komplexität
        - Gestaltprinzipien
    - Evaluation
    - Implementierung
    - Usability nach Auslieferung
- Usability Evaluation
    - Review durch Experten
    - Evaluationsregeln
    - Evaluationsarten
    - Usability Test
    - Testarten
    - Vorbereitung
    - Einführung
    - Durchführung
    - Evaluation
    - A/B-Test
- Programmierprinzipien
    - Solid
        - (S)ingle Responsibility
        - (O)pen Closed Principle
        - (L)iskov Substitution Principle
        - (I)nterface Segregation Principle
        - (D)ependency Inversion Principle
    - Tell, don’t ask
        - Prozedurale Vorgehensweise
        - Objektorientierte Vorgehensweise
    - Kiss (Keep it simple, stupid)
    - SLAP (Single Level of Abstraction Principle)
    - GRASP
        - Low Coupling
        - High Cohesion

Unit-Testing

Fehler

- Fehlerursachen: Falsche Einschätzung von Risiken, Fehlende Tests
- Fehler kosten Ressourcen
- Kosten steigen mit der Zeit

Tests

- Verpflichtung zum Testen

- Tests schützen bestehende Funktionen
- Orientierungshilfe und Dokumentation
- Neuere Entwicklungsmethoden: Test First, Test Driven Development
- Nutzen statt Aufwand
- Tests unterscheiden zwischen zufälliger und gewollter Funktionalität

Testarten

- Akzeptanztest
  - **Test des kompletten Systems**
  - Realistische Umgebung, Steuerung durch Mittel des Benutzers, Absegnung durch Auftraggeber
  - **Ziel**: Echte Benutzerszenarien
- Integrationstest
  - Nur relevante Teile des Systems werden gestartet
  - Stellvertreter für nicht relevante Teile
  - Testframework
  - **Zusammenspiel der Komponenten sichern**
- Komponententest
  - Nur relevanter Teil des Systems wird gestartet
  - Alle anderen Teile durch Stellvertreter ersetzen
  - Testframework
  - **Funktionalität einzelner Komponenten sicherstellen**
- Performancetest

Was ist eine Komponente?

- Meist Klasse
- Ersetzen der Abhängigkeiten durch Stellvertreter „Mocks“
- Mock: Minimale notwendige Funktionalität

xUnit-Testframework

- Vorlage für Unit-Tests
- Trennung zwischen Test- und Produktivcode
- Für fast alle Sprachen existiert Implementierung

Aufbau eines Tests

- Arrange: Initialisieren der Testumgebung (String def.)

- Act: Ausführen des zu testenden Codes (Lowercase String)
- Assert: Überprüfen des Ergebnisses (Is der String lowercase?)

JUnit

- Quasi-Standard in der Java-Entwicklung

Überprüfungen

- asserts
  - Überprüfung auf Gleichheit, Referenz, Wahrheit, Null
  - assertEquals("blub", result);
- Matcher
  - Möglichkeit, um weitere Überprüfungen hinzuzufügen
  - Bestehen aus:
    - Überprüfung eines Wertes mit einem erwarteten Wert
    - Beschreibung, welcher Wert erwartet wird („start at…“)
    - Beschreibung, welcher Wert überprüft wurde („started at…“)
  - Zahlen auf größer/kleiner, von Objekten auf Gleichheit, von Listen und ähnlichem
  - assertThat(list, contains(„element“));
  - Können verschachtelt werden
  - assertThat(number, is(lessThan(0)));

Vorteile	Nachteile
Verbessern die Lesbarkeit des Tests	Implementierung braucht Zeit
Sprechendere Fehlermeldungen	
Wiederverwendbarkeit	

Exceptions und Spezialfälle

- Exceptions:
  - Gewisse Exceptions können ebenfalls erwartet werden (Annotation möglich, oder Assert)
- Gleitkommazahlen lassen sich nicht exakt überprüfen
  - Genauigkeit kann angegeben werden
  - assertThat(value, closeTo(1.0d, 1E-2))
- Arithmetische Spezialfälle
  - Ganzzahlen kennen keinen Overflow

    - Kleinster Wert von Datentypen haben unterschiedliche Semantik
    - Double kennt kein DivisionByZero

Ergebnis eines Tests

- Success
    - Testmethode erfolgreich durchlaufen
    - Keine Assertion fehlgeschlagen
- Failure
    - Assertion ist fehlgeschlagen
- Error
    - Unerwarteter Fehler

Eigenschaften guter Tests – ATRIP

- Automatic
    - Minimale Anforderung bei der Ausführungo
    - Einfach ausführbar (1 Knopf), automatisch (kein Input), selbst-überprüfbar (bestanden/fehlgeschlagen)
- Thorough (Vollständig)
    - Alles Notwendig überprüfen
    - Iteratives Vorgehen (bei Fehler ergänzen)
    - VFehler klumpen: Bei Fehler auch Umgebung testen

Repeatable

- Beliebig wiederholbar -> gleiches Ergebnis
    - Plattform, Zeit, Zufall, Multithreading kritisch
- Tests, die ohne Änderungen fehlschlagen -> Fehlerhafte Tests

> Zufall muss gesteuert werden. Statt random() feste Zahl wählen.

Independent

- Keine Abhängigkeit zu anderen Tests (keine Reihenfolge)
- Tests sollen auf Aufgabe fokussiert sein

Professional

- Tests sollen qualitative hochwertig sein (kein Doppelcode etc.)

- Keine unnötigen Tests
- Teil der Dokumentation

Mock-Objekte

Verhalten im Test

- Reduzieren Abhängigkeit zu anderen Komponenten
- Stellvertreter für Objekte
- Mocks können manuell erstellt werden (Großer Aufwand) oder durch Mocking-Frameworks
- Arrange: Konfiguration der Mocks
- Act: Verwendun der Mocks
- Assert: Überprüfen der Mocks
- Beispiel: HW-Gerät verlangsamt Test, HW soll nicht getestet werden -> Mockup durch Interface abilden

Schwierigkeiten

- Verwendung von statischen Methoden (-> Dependency Injection)
- Tiefe Abhängigkeiten erschweren Mocking (-> Lose Kopplung)
- Vorsicht vor Testen von reinem Mock-Verhalten?

Code Coverage

- Tests testen
  - Testabdeckung
  - Temporäre Probleme einbauen
  - SW, die Produktivcode ändert (Mutation Testing)
- Coverage: Wie viel Code wurde während Test durchlaufen?
- Branch Coverage: Misst die Anzahl der durchlaufenen Pfade (if, else etc.)
- Line Coverage: Misst die Anzahl der durchlaufenen Quellcode-Zeilen
- Branch/Line -> unterschiedliche Aussagekraft, keine Aussage über Funktion, deuten auf
  potentiell problematische Bereiche hin

Test First

- Klassisch: Funktion planen -> Programmieren -> Refactoring -> Testen -> Fehler beheben

- Test First: Funktion planen -> Tests -> Programmieren -> Refactoring
- TDD: Tests in kleinen Schritten entwickeln (bis fehlschlägt) -> Funktion ändern bis Test erfüllt
- Kerngedanke
  - Tests > Produktivcode
  - Entwicklungsende, wenn Tests erfüllt -> Minimal notwendiger Code
  - Test als Wegweiser

Vorteile	Nachteile
Volle Testabdeckung	Testen wird zur Pflicht
Fehlerrate sinkt	Ungewohntes Vorgehen / Einarbeitung
Angst existierende Fnkt. zu ändern, sinkt	Aufwand zur Implementierung höher
Automatische Spezifikation/Dokumentation	Ggf. schwierige Anwendung
Weniger Produktivcode	

Umgang mit bestehendem Code

- Bestehender Code schwer testbar (Legacy Code)
- Legacy Code nicht fürs Testen entwickelt (viele Abhängigkeiten)
- Kann schrittweise weiterentwickelt werden (absichern, isolieren)

Testen auf der grünen Wiese

- Neue Projekt -> von Anfang an testbar
- Feature -> „Ende-zu-Ende-Test“ definieren, bei Erfüllung ist Feature fertig
- Einzelne Schritte -> Unit-Tests

Refactoring

Was ist Refactoring?

- Bestehenden Code durchgehen: Code Review
- Intention/Funktion nachvollziehen
- Testabdeckung wird überprüft
- Code wird umgestaltet (Verhalten & Schittstellen bleiben gleich, neues Wissen einbauen)

- Ziel: Codequalität verbessern (lesbarer, flexibler nutzbar, Struktur verbessern)

Warum sollte man Refactoring verwenden?

- Design der SW wird verbessert
- SW wird wartbarer
- SW wird einfacher verständlich
- Fehler werden einfacher gefunden
- Neue Funktionen kann schneller entwickelt werden
- Optimierung von SW

Wann sollten wir Refactoring verwenden?

- Bestandteil der Entwicklung -> kontinuierlich
- Regel: Three strikes and you refactor
    - 1. Implementieren -> neue Funktionalität
    - Ähnliche Funktion? -> Code kopieren
    - Erneut Ähnliche Funktion? -> Code refactorn und wiederverwendeno
- Code Review
- Vor Hinzufügen neuer Funktionalität
- Beim Beheben eines Fehlers
- Lesbarkeit von SW verbessern
- Optimierung, hält Geschwindigkeit konstant

Wann wird Refactoring schwierig?

- DB-Schemata schwieriger änderbar
- Änderung von Schnittstellen
- Zentrale Designentscheidungen
- Aufwand zu groß? -> Komplett neu entwickeln

Auswirkungen auf das Design

- Funktion, wird während der Implementierung verstanden
- überarbeiten der Software kann Komplexität reduzieren
- Software lässt sich schnell ändern => Design dynamisch
- Gegensatz: Ingenieur

Nachteile von Refactoring

- Refactoring kostet Zeit
- Verschlechtert die Performance

Code Smells

- Code kann stinken, Stärke des „Gestanks“ schlecht messbar
- Deuten auf verbesserungswürdige Stellen im Code hin und welche die die Entwicklung behindern

Duplicated Code

- Doppelt vorhandener Code
- Gleiche Code-Struktur an mehr als einer Stelle
- Auseinanderdriften mehrere Stellen
- Lösung: Gemeinsamen Code auslagern

Long Method

- lange Methoden
- Je länger -> schwerer verständlicher
- OO: viele Methoden delegieren
- Semantische Distanz zwischen Code und Name der Methode
- Lösung: Auftrennen der Methode (gute Namen, Kommentare, Schleifen, ifs sind gute Stellen)

Large Class

- Große Klasse
- Anzahl Instanzvariablen, Methoden, Präfixe, Suffixe bei Variablen
- Klasse: Zu viel Verantwortung
- Lösung: Unterteilen der Klasse

Shotgun Surgery

- Flickenteppich Änderung
- Kleine funktionale Änderung -> Anpassung vieler Stellen
- Lösung: Umstrukturierung des Codes

Switch Statements

- Switch Statements: oft gleiche Switchs, wenige Nahtstellen zum Auftrennen, Fehler durch
- vergessenes Break
- Lösung: OO

Code Comments

- Schlechter Code muss erklärt werden
- Lösung: Kommentar ist guter Indikator zur Trennung von Methoden -> Intention des
- Kommentars als Methodenname

Refactorings

Extract Method

- Problem: Zusammenhängendes Stück Code kann extrahiert warden
- Lösung: Code in eigene Methode auslagern, passenden Namen wählen, Code freingranular aufteilen
- Verbesserung: Bessere Strukturierung des Codes
  Hilft gegen: Code Comments, Duplicated Code, Long Method

Rename Method

- Problem: Methodenname passt nicht zum Inhalt der Methode
- Lösung: Methodenname ändern
- Verbesserung: Code lesbarer/verständlicher
- Hilft gegen: Code Comments

Replace Temp with Query

- Problem: Ergebnis einer Berechnung wird temporär in einer Variablen gespeichert
- Lösung: Berechnung des Wertes in eine Methode auslagern
- Verbesserung: Extrahierung, Schreibzugriffe auf Variablen werden sichtbar, Wert der
- Berechnung wird nicht zwischengespeichert
- Hilft gegen: Long Method

Replace Conditional with Polymorphism

- Problem: Verhalten mit Konditionalstrukturen und einer Typ-Kodierung gesteuert
- Lösung: Verhalten der einzelnen Pfade in abgeleitete Klasse überschreiben, Basismethode abstrakt

- Verbesserung: Software besser gekapselt, dynamisch erweiterbar
- Hilft gegen: Switch-Statements

Replace ErrorCode with Exception

- Problem: Fehlerwerte werden zurückgegeben
- Lösung: Statt Fehlerwert lieber Exception
- Verbesserung: Klare Definition von Fehlern, bessere Steuerung, Code verständlicher

Replace Inheritance with Delegation

- Problem: Funktionalität in abgeleiteter Klasse nicht brauchbar
- Lösung: Instanzvariable mit dem Type der Oberklasse anlegen
- Verbesserung: Klarer def. Schnittstellen, Trennung zw. eigener u. vorhandener Funktionalität

GUI Blooper

Was sind GUI-Blooper?

- Allgemeine GUI-Design-Fehler
- Über 70 GUI-Blooper

Durch Blooper verursachte Probleme

- Verwirrung beim Nutzer
- Unnötiger Zeitaufwand
- Datenverlust

Gründe für Blooper

- Fehlende Zeit, Wissen, Ressourcen
- Arten von Bloopern
  - GUI-Komponenten
  - Navigation
  - Texte
  - Design und Layout
  - Interaktion
  - Management
- Antwortverhalten

GUI Komponenten

- Umgang mit GUI-Komponenten
- Aussehen > Wahl der Komponenten
- Fehlende Unterstützung durch GUI-Toolkit
- Fehlendes Fachwissen
- 2 Kategorien
    - Falsche Komponente
    - Falsche Verwendung von Komponenten
- Beispiele:
    - Verwirrende Checkboxen/Radiobutton (Einzelner Radiobutton, Checkbox als Radiobutton)
    - Textfelder für beschränkten Input (Datumsfeld, Radiobutton, …)

Navigation

Navigation allgegenwärtig, wichtig für Orientierung
Aktueller Ort, Vorheriger Ort, Mögliche nächste Orte, Entfernung zum Ziel
Beispiele:

- Gleicher Titel für unterschiedliche Fenster
- Zu viele Ebenen von Dialogboxen

Text

- Viel Text in GUI
- So wenig wie möglich, so viel wie nötig
- Visualisierung oft besser
- Kategorien:
    - Unkommunikativ
    - Entwicklerzentriert
    - Fehlleitend
- Beispiele:
    - Inkonsistente Terminologie
    - Speaking Geek: Entwicklersprache für Nutzer unverständlich

Design und Layout

- Design, Farben, Layout
- Amateurhaft vs. Professionell
- In der Regel einfach korrigierbar
- Beispiele:

    - Leicht übersehbare Informationen
    - Radiobuttons zu weit auseinander (-> Groupboxen)

Interaktion

- Schwierig zu erkennen/beheben
- Decken größere Bereiche ab
- Arten:
    - Ablenkung von eig. Aufgabe
    - Unnötige Abläufe
    - Gedächtnis des Users belasten
    - Kontrolle des Users entziehen
- Beispiele:
    - Unnötige Beschränkungen (Zeichenbeschränkung etc.)
    - Sinnlose Auswahlmöglichkeiten (HTTP1 oder HTTP2?)

Management

- UI unterschätzt durch Management
- UI überschätzt von Management
- Beispiele:
    - UI als niedrige Priorität
    - Anarchische Entwicklung
    - Programmierer bekommen schnellsten Computer

Antwortverhalten

- Benutzer immer ungeduldig
- Intransparenz der Aktion
- Mehrfaches Ausführen
- Beispiele:
    - Lag zerstört Hand-Augen-Koordination
    - Anwendung zeigt nicht, dass sie beschäftigt ist
    - Lange Operationen ohne Möglichkeit abzubrechen

Responsiveness

- Responsivness != Performance
- Ressourcen sind limitiert
- UI ist ein real-time Interface
- Verzögerungen sind nicht gleich

- Aufgaben nicht in der Reihenfolge in der sie erscheinen
- SW braucht nicht alle Aufgaben machen, die gefordert wurden

Vermeidung von langem Antwortverhalten

Timely Feedback

- Benutzereingaben direkt bestätigen

Parallel Problem Solution

- Prioritäten vergeben (Usereingaben > Update)
- Vorarbeiten

Queue Optimization

- Unkritische/Unnötige Aufgaben verzögern/entfernen

Dynamic Time Management

- Bearbeitungszeit berechnen
- Abarbeitung der Warteschlange ändern

UI Entwicklung

Sinnvoller Designprozess

- Workflow (Interface Designer) -> GUI (IF Designer / Grafiker) -> Businesslogik (Programmierer)

Definition von Usability

- Usability eines Produktes ist das Ausmaß, in dem es von einem Benutzer verwendet werden kann, um Ziele in einem Kontext effektiv, effizient und zufriedenstellend zu erreichen.

Wesentliche Elemente

- Benutzer
- Ziele
- Kontext

Wesentliche Messgrößen

- Effektivität
- Effizienz
- Zufriedenheit

5Es

- Effective: Vollständigkeit/Genauigkeit
- Efficient: Geschwindigkeit
- Engaging: Zufriedenheit
- Error tolerant: Vermeidung von Fehlern
- Easy to learn: Unterstützung bei erster Bedienung

User Centered Design Process

- Ziel: Benutzerfreundliche Oberfläche
- Analyse -> Design -> Implementierung -> Deployment
- Während Design und Implementierung auch Evaluation

Analyse

- Aufgabe: Informationen sammeln über
  Benutzer, Aktivitäten/Ziele, Umfeld
- Ziel: Mentales Modell des Produkts
  Folie 9, 10, 12

Informationsquellen

- Marktforschung
- Interne Schulungsunterlagen
- Benutzer beobachten/interviewen

Informationen zum Benutzer

- Wichtigste Komponente: Benutzer
- Eigenschaften wie Name, Alter, Geschlecht, Bild, Job, Ziele, Domänenwissen, Umfeld
- Grundlage für Tester
- Vorteile:
  - Konkretes Bild eines Benutzers
  - Bessere Identifikation mit dem Benutzer

Mentales Modell

- Grundlage/Inhalt

    - Probleme des Nutzers
    - Ziele des Nutzers
    - Daten, die der Nutzer verarbeitet
- Zusammen mit dem Kunden erstellen
- Verwendung der Begriffe des Benutzers
- Aufgabenspezifisch ohne UI-Begriffe
- => Zuerst die Funktion, dann das Aussehen

Objekt-Aktions-Analyse

- Welche Objekte und Aktionen gibt es?
- Beziehungen und Hierarchien festhalten
- Keine Implementierungsdetails

Regeln an ein mentales Modell

- Schlichtheit/Einfachheit
- Vertrautheit
- Flexibilität
- Sicherheit
- Affordances

Lexikon

- Wörterbuch für Begrifflichkeiten
- Verständlich für den Benutzer
- Gleiche Aktion -> Gleicher Begriff
- Kompakt
- => Konsistenz zwischen SW und Doku

Szenarios

- Beschreiben Abläufe einzelner Aktivitäten
- Blaupause für Usability-Test

Gemeinsame Diskussionsgrundlage

- Für alle einsehbar
- Blaupause für erste Implementierung

Vorteile

- Ziel / Aktivitäten bezogen: Festlegung von Relevanz und Beziehungen
- Wichtigkeit: Ordnung der einzelnen Komponenten
- Konsistenz: Definition allg. Aktionen, Verwendung einheitlicher UI, Einfache Korrekturmög.

Design

- Aufgabe: Umfang und Aussehen der SW festlegen
- Anzahl der Features, Wichtigkeit der Features, Bedienkonzepte (Workflow)
- Ziel: Prototyp für Usability Tests und weitere Entwicklung

Umfang der Software

- Sicht des Benutzers Grundlage für die UI
- Verwende natürliche Abläufe und Begriffe
- So wenig Beschränkungen wie möglich
- Anzahl Features vs. Komplexität
- => Sichtweise des Benutzers auf die Aufgabe zählt

Anzahl Features versus Komplexität

- Standardwerte
- Templates
- Wizards
- Schrittweise Offenlegung
- Generische Befehle
- Aufgabenspezifisches Design
- Anpassungsfähigkeit
- Standardfall einfach erreichbar
    - Arten: Anzahl Benutzer, Aufrufhäufigkeit
- UI sorgfältig planen
- Benutzer die Kontrolle überlassen
- Minimale Änderung bei neuen Daten

Gestaltprinzipien

- Gute Beschreibung menschlicher Wahrnehmung
- Gute Richtline für UI-Design
- Grundlage für Bedienkonzepte
- Sind allgemeingültig
- Kombination ist möglich

- Erkenntnis über unterwünschte Gruppierung und Fokussierung
- Nach UI-Entwurf Prinzipien überprüfen und unerwünschte Effekte entfernen
- Prinzipien:
  - Proximity: Nähe gruppiert
  - Similarity: Ähnlichkeit gruppiert
  - Closure: Offene Objekte werden geschlossen wahrgenommen
  - Figure/Ground: Einteilung in Voder- und Hintergrund (Vordergrund: Fokus)
  - Common Fate: Gemeinsame Bewegung wirken gruppiert

Evaluation

- Paper Prototyping
- Expert Review
- Usability Testing

Implementierung

- Antwortverhalten
- Gestaltungsrichtlinien
- GUI-Tests

Usability nach Auslieferung

- Feldtest
- Logoanalyse
- Langzeitstudien

Usability Evaluation

Review durch Experten

- Überprüfung durch Usability Experte / Domain Experte
- Prüfung basierend auf einfach Regeln

Evaluationsregeln

- Sichtbarkeit des Systemstatus (aktuell)
- Unterschiede zw. Realität und System (Sprache, Abfolge)
- Konsistenz / Standards

- Flexibilität, Effizienz
- Gedächtnis des Nutzers entlasten
- Minimalistisches Design
- Benutzer einen Ausweg lassen
- Fehlervermeidung
- Unterstützung bei Fehlern
- Hilfe und Dokumentation

Evaluationsarten

- Formell:
  - Experten -> Berichte -> Zusammenfassung der Berichte
  - Klassifizierung der Probleme
- Informell
  - Teammitglied überprüft
  - Informelles Meeting

Usability Test

- Überprüfung durch echte Nutzer
- Zeit für Korrekturen einplanen
- Ziel: Information, soziales Ziel

Testarten

- Formativ: klein, iterativ, während Entwicklung, für ein Ziel -> günstig, schnell
- Summativ: umfangreich, vor Auslieferung

Vorbereitung

- Benutzerprofile erstellen -> Auswahl der Benutzer
- Szenarien/Ziele definieren (Szenario beschreibt „Suche nach Informationen“)
- Umfang definieren
- Teilnehmer rekrutieren
- Zeitraum festlegen
- Testablauf für Beteiligten beschreiben•
- Labor / Unterlagen vorbereiten

Einführung

- Teilnehmer begrüßen
- Angenehme Atmosphäre schaffen
- Beteiligte vorstellen
- Räumlichkeit zeigen
- Szenarien / Ziele erklären
- Verhalten während Test besprechen
- Produkt wird getestet, nicht Teilnehmer
- Lautes Denken
    - + Besseres Verständnis
    - - Ungewohnt, passende Umgebung notwendig

Durchführung

- Teilnehmer soll Lösung selbst finden
- TN bestimmt Tempo
- Genügend Pausen
- Klare Aufteilung zw. Moderator / Beobachtern
- Produktexperte für Nachfragen
- Bei Verwirrung nachfragen
- Teilnehmer verabschieden
- Testumgebung
    - Testutensilien
    - Eigenes Labor
    - Basisaufwand gering, größter Nutzen
    - Teuer, großer Platzbedarf
    - allg. nutzbarer Raum
    - Günstig, wenig Platz notwendig
    - Höherer Basisaufwand
- Feldtest: pot. überall
- Reale, gewohnte Umgebung
- Ablenkung, höhere Kosten, Umgebung nicht festlegbar
- Remotetest: örtlich getrennt
- Synchron (verbunden via. Video): vielfältig, günstig, zeitsparend, schwierig, Setupaufwand, allg. Prob. von Remote
- Asynchron (Vordef. Fragen, Aufzeichnung): mehr Teilnehmer, vgl. mit Konkurrenzprod., kein Audio/Blickkontakt, teuer
- Testutensilien
    - Basic: Möbel, Geräte
    - Nice to have: Kamera, Mikrofon, …

    - Special: Eye-Tracker, …
- Auswahlkriterien
    - Budget
    - Ressourcen
    - Größte des potentiellen TeilnehmerkreisesHäufige Fehler
    - Verwendung von Wörtern der UI
    - Beeinflussung des TN
    - Erzeugung von Stress
    - Benutzer gibt sich die Schuld am Fehler

Evaluation

- Hauptfragen: Was wurde gesehen, was bedeutet es, wie damit umgehen?
- Evaluation durch versch. Personen
- Einteilung der resultierenden Aktionen: Dringend, lokal, global

A/B-Test

- Redesign vs. bestehende SW
- Unklar, welcher besser -> Benutzer in Gruppen teilen und je eine Version geben
- Usability messen und statistische Signifikanz beachten

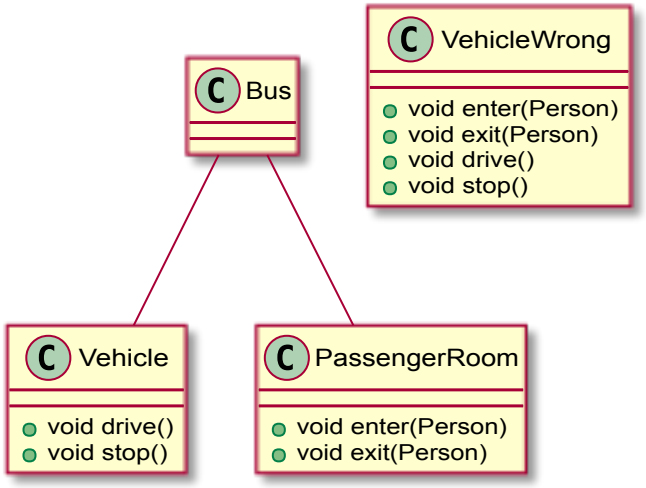
Programmierprinzipien

- sind Leitfaden
- Verantwortung festlegen

Solid

(S)ingle Responsibility

- Klasse sollte nur einen Grund oder Ursache haben, sich zu ändern
- jede Klasse nur eine Zuständigkeit
- eine Klasse erhält Achsen, auf der sich Anforderungen ändern können
    - jede Zuständigkeit-> neue Achse, nur eine Achse pro Klasse



(O)pen Closed Principle

Elemente der Software wie Klassen, Module und Funktionen sollten

- offen für Erweiterung sein
- geschlossen für Änderungen sein

Erweiterung nur über Vererbung bzw. Implementierung von Interfaces (optimal)

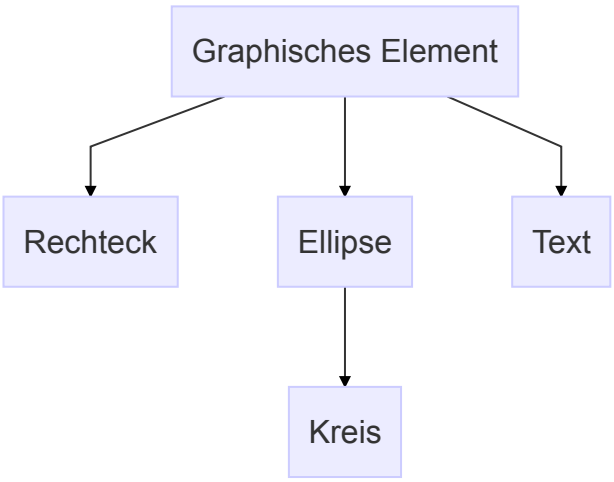
> bestehender Code wird nicht geändert

- Abstraktionen fördern die Erweiterbarkeit
- Software nie immun gegen Änderungen

(L)iskov Substitution Principle

- Abgeleitete Typen müssen schwächere Vorbedingungen haben
- Abgeleitete Typen müssen stärkere Nachbedingungen haben
- Beispiel Quadrat($width^2$) erbt von Rechteck($width * height$)

> Wenn sich das Objekt so verhält, wie sein Oberklasse

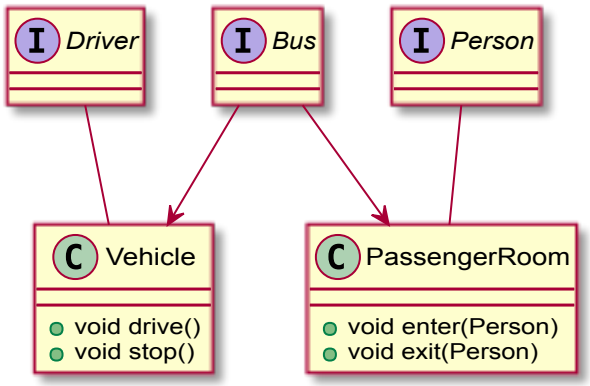


(I)nterface Segregation Principle

Anwender sollen nicht von Funktionen abhängig sein, die sie nicht brauchen

> Übergebe User nur Interface mit Funktionen, die er benötigt

- Typen implementieren meist mehrere Interfaces



(D)ependency Inversion Principle

High-Level Module von Low-Level Modulen abhängig

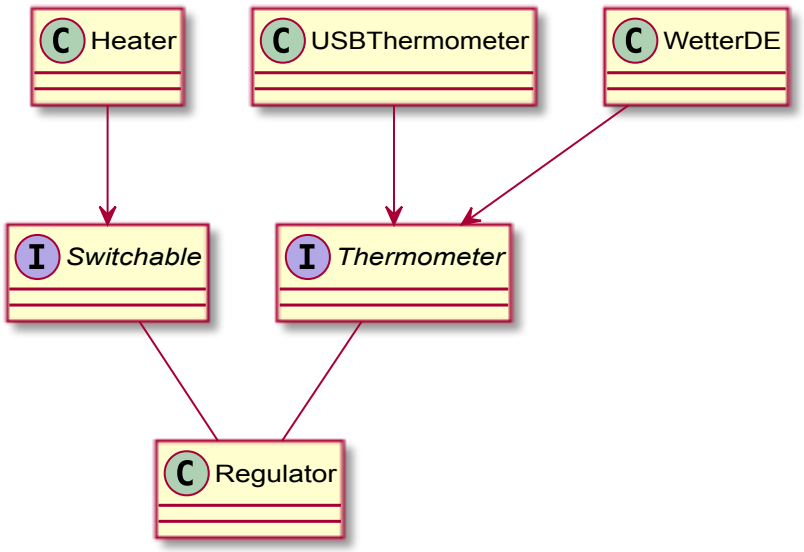
> Änderung in Low-Level Implementierung ändert High-Level Modul

- schlecht
- besser => High-Level Modul von Abstraktionen abhängig
- Abstraktionen nicht von Details abhängig
- Details abhängig von Abstraktionen

> - Regeln durch High-Level Module vorgeben

> - Low-Level implementiert Regeln
> - High-Level können wiederverwendet werden (bilden Framework)

Beispiel UML Klassen Diagramm



- Klassen sollten nur abstrakte Klassen oder Interfaces ableiten und implementieren
- Variablen und Members sollten eine abstrakte Klasse oder Interface als Typ haben
- nur abstrakte Methoden implementieren
- beim Initialisieren der Anwendung werden Instanzen konkreter Klassen erzeugt

Tell, don’t ask

- Prozeduraler Code kappelt sich stark an andere Elemente
- Kommandos an Objekte besser als Abfragen
- holt sich erst Informationen, entscheiden Datenbankschema

Prozedurale Vorgehensweise

- Status eines Objektes Abfragen
- Entscheidung treffen

> führt zu zentraler Businesslogik

Objektorientierte Vorgehensweise

- Element etwas ausführen lassen
- Objekte Experten ihrer internen Informationen
- Objekt hat alle Informationen, um eine Entscheidung selbst zu treffen

> führt zu verteilter Businesslogik

Kiss (Keep it simple, stupid)

Herkunft in der US Navy 1960

- einfache Systeme arbeiten am besten
- Komplexität unter allen Umständen vermeiden
- Linux Arch
- Komplexität erhöht Chance einen Fehler zu machen

SLAP (Single Level of Abstraction Principle)

- Prinzip des einfachen Abstraktionsniveau
- keine Vermischung von Arbeit und Delegation
- keine Vermischung von DB und Businesslogik
- Fördert Wiederverwendbarkeit

GRASP

- General Responisbility Assigment SW Patterns
- Basis-Prinzipien auf denen Entwurfsmuster aufbauen
- Low Representational Gap (LRG) minimieren
  - Lücke zwischen Domänenmodell und Implementierung
- Zuweisung von Zuständigkeiten (2 Typen: Ausführung, Wissen)

Low Coupling

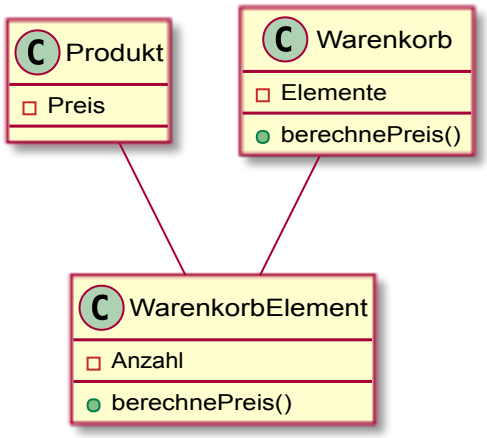
- Geringe Kopplung, Abhänigigkeit zw. Objekten
- Leichter änderbar, testbar, wiederverwendbar, verständlicher
- Bsp: Impl. von Interfaces, Vererbung, gemeinsame Dateien, Locks durch Threads

High Cohesion

- Zusammenhalt einer Klasse: Semantische Nähe der Elemente
- Einfacher, verständlicher, wiedervendbarer
- Schwer bestimmbar, ggf. durch Anzahl Verwendungen, Anzahl Attribute

Information Expert

- Zuweisung einer Zuständigkeit zu einem Objekt
- Objekte sind zuständig für Aufgaben über die sie Informationen besitzen
- Kapselung von Infos, leichtere Klassen <-> ggf. Problem mit anderen Prinzipien



Creator

- Wer ist für Erzeugung eines Obj. Zuständig?
- Wenn das Objekt zu jedem erstellen Objekt eine Beziehung hat (z.B. Komposition, wenn a Teil von B ist [Raum, Haus])
- Verringert Kopplung

Indirection

- Indirektion/Delegation, kann Syteme oder Teile voneinander entkoppeln
- Mehr Freiheitsgerade als Vererbung
- Komposition verschiedener Objekte möglich
- Bsp: Objekt nutzen statt davon erben

Polymorphism

- Behandlung von Alternativen abhängig von einem konkreten Typ
- Methoden erhalten je nach Typ andere Implementierung
- Vermeidung von Fallunterscheidung
- Abstrakte Klasse, Interface als Basistyp
- Polymorphe Methodenaufrufe erst zur Laufzeit gebunden
- -> Entwurfsmuster Strategie
- Beispiel: Steuer-Interface, Deutschland, Frankreich-Klasse
- Erweiterbar, bestehendes muss nicht geändert werden, extrahierung von Frameworks vereinfacht

Controller

- Verarbeitung von einkommenden Benutzereingaben
- Koordination zwischen UI und Logik
- Delegation zu anderen Objekten

- Zustand der Anwendung kann in Controller gehalten werden
- Arten
    - System Controller: Controller für alle Aktionen
    - Use Case Controller: Controller pro Use-Case,

Pure Fabrication

- Reine Erfindung, reine Verhaltens- / Arbeitsklasse ohne Bezug zur Domäne (möglichst selten!)
- Trennung zw. Technik und Domäne
- Wiederverwendbar, High Cohesion

Protected Variations

- Sicherung vor Variation
- Kapselung versch. APIs hinter einheitlicher API
- Polymorphie, Delegation als Schutz
- Bsp: OS (HW), SQL (DB)

DRY (Don’t Repeat Yourself !)

- wiederhole dich nicht
- Anwendbar:
- Datenbankschema
- Testpläne
- Buildsystem
- Dokumentation
- Gegenteil:
- WETYAGNI (You ain’t gonna need it)du wirst es nicht brauchen

YAGNI

- You ain’t gonna need it (Du wirst es nicht brauchen)
- Unnötige Features erhöhen Komplexität, binden Ressourcen
- Eigene Ideen -> schwer objektiv betrachtbar
- Frameworks sinnvoll, wenn sie aus dem Projekt heraus entstehen, nicht wenn sie durch
  spekulatives Programmieren entstehen
- Kommunikation zw. Entwicklung u. Kunde wichtig

Conway’s Law

- Kommunikationsstruktur findet sich in Code wieder
- Kommunikationsschnittstellen = Modulschnittstellen im Code
- Müssen zum Produkt passen…
- Bei Neuausrichtig des Produkts -> Kommunikationsstruktur anpassen
- Beispiel: Konzernwebseiten spiegeln Org. wieder statt Bedürfnisse des Kunden

DevOps

- eine **Bewegung** mit dem Ziel **Time-To-Market** einer **Änderungseinheit** zu reduzieren, bei gleichzeitiger Gewährleistung **hoher Qualität**
- durch Anwendung des **Lean-Prinzip** auf den gesamten **Software-Wertstrom**

Warum

- Dev **schnell Veränderungen umsetzen**
- Ops (Administrator) sollen **Sicherheit und Stabilität** der Systeme gewährleisten

Lean

Philsophie, mit Ziel, einen **Prozess** durch die **Eliminierung von Verschwendung kontinuierlich** zu verbessern und dabei die **Bedürfnisse der Kunden** als **Ausgangspunkt allen Handelns** sieht

Verschwendung erkennen:

- Materialbewegung
- Bestände::
- Bewegung
- Wartezeiten
- Verarbeitung
- Überproduktion
- Korrekturen und Fehler

Verschwendung beseitigen:

Pull Prinzip:

- es wird nur produziert:
    - **was** der Kunde will

    - **wenn** der Kunde es will

Entwurfsmuster

- Muster beschreiben wiederkehrende Probleme

Nutzung von Entwurfsmustern

- Vermittlung von Wissen auf abstraktem Niveau
- Ausprägung einer höherwertigen Sprache in OOP
- Helfen komplexe SW-Systeme zu beherrschen

Gliederung von Entwurfsmustern

- Nach Zweck
    - Erzeugungsmuster
    - Strukturmuster
    - Verhaltensmuster
- Nach Geltungsbereich
    - Klassenebene: Beim Kompilieren festgelegt
    - Objektebene: Bei Laufzeit festgelegt

Erzeugungsmuster

- Trennen Erstellung von Verwendung von Objekten
- Instanzen werden einfacher ersetzbar für anderes Verhalten
- System unabhängig von Implementierung der Objekte
- Kapseln Wissen

Strukturmuster

- Kombinieren Klassen u. Obj., um größere Strukturen zu schaffen
- Kombination von mehreren Interfaces
- Kombination von Funktionalität zur Laufzeit
- Übersetzung von einem zum anderen Interface
- Sparen von Ressourcen

Verhaltensmuster

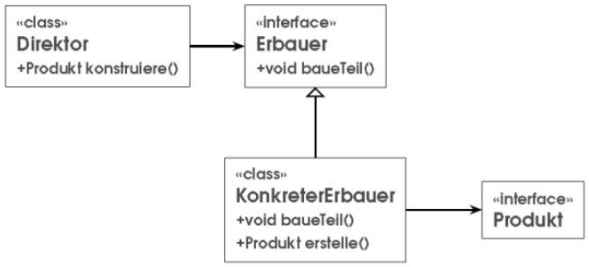
- Zuweisung von Verantwortlichkeiten an Obj.

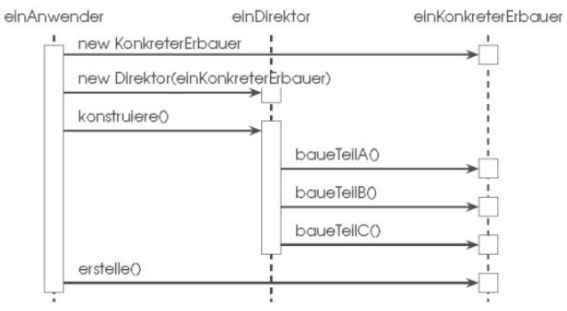
	Erzeugungsmuster	Strukturmuster	Verhaltensmuster

- Austausch von Verhalten
- Kommunikation zw. Obj.
- Steuerung des Kontrollflusses einer Anwendung zur Laufzeit

	Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Klasse	Fabrikmethode	Adapter	Schablonenmethode
Objekt	Erbauer	Kompositum Dekorierer	Beobachter

- **Trennung der Erstellung von ihrer Repräsentation von (komplexen) Objekten**
- Gleicher Erstellungsprozess -> ggf. unterschiedliche Repräsentationen
- Wiederverwendung einer komplexen Logik zur Umwandlung von Objekten
- Erzeugungslogik für versch. Formate von Konvertierungslogik trennen
- **Schrittweise Erzeugung von komplexen Produkten**
- **Wiederverwendung der Erzeugungs- bzw. Konstruktionslogik voneinander**
- **Vereinfacht die Erweiterbarkeit**
- Akteure
- Erbauer: Schnittstelle zur Erstellung der Teile eines Produkts
- Konkreter Erbauer: Erzeugt, verwaltet und setzt zusammen versch. Teile d. Prod., Implementiert Erbauer, Möglichkeit zur Erzeugung d. Prod.
- Direktor: Konstruiert m.H. des Erbauers ein Prod. o Produkt: Komplex erzeugtes Produkt



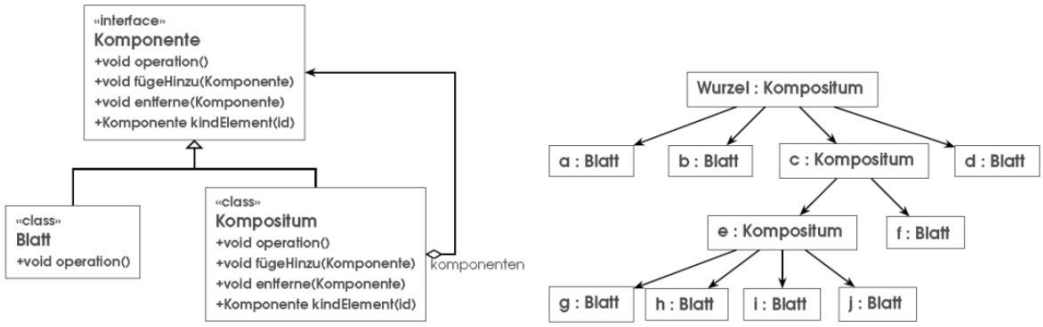


Auswirkungen

- Interne Repräsentation des Prod. kann variieren o Genaue Kontrolle über Konstruktionsprozess
- Trennung von Code zur Erstellung und Repräsentation

Kompositum

- Setze Obj. zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu bilden
- Kombination einfacher Elemente zur Erzeugung komplexer Strukturen
- Anwender soll Elemente (die etwas ausführen) und Komposita (Container, die etwas aufnehmen) nicht unterscheiden müssen
- Akteure
  - Anwender: Manipuliert Objekte im Kompositum nur über Interface
  - Komponente: Interface der Obj. im Kompositum, für Verwaltung der Kinder, Standardverhalten für alle Fälle
  - Blatt: Keine Kinder, definiert Verhalten einfacher Objekte
  - Kompositum: Def. Verhalten für Obj. mit Kindern, verwaltet Kinder, Implementiert Verhalten bezogen auf Kinder

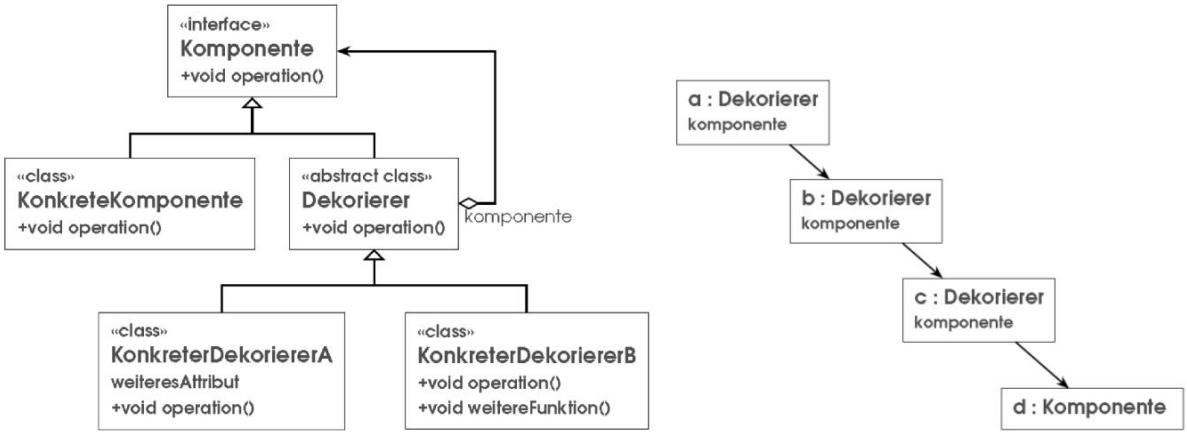


- Auswirkungen
- + Einfache Elemente können beliebig zusammengebaut werden -> Rekursive Verschachtelung
- + Vereinfacht Logik beim Anwender
- + Neue Komponenten können einfach def. werden
- - Design zu generell
- Transparenz, Typsicherheit
- Folie 39 – 42?

Dekorierer

- Dynamische Zuweisung einer weiteren Verantwortung zu einem Objekt
- Erweitern eines Objekts mit Funktionalität
- Flexible Alternative zu Objekthierarchien
- Einhaltung einer flachen Objekthierarchie
- Leichtgewichtig, Instanzreich
- Verschachtelung von Obj. zum Hinzufügen von Funktionalität bietet mehr Freiheiten als Vererbung

- Zusatzfunktion bleibt transparent
- Wenn Ableitung einer Klasse zu komplex ist
- Akteure
- Komponente: Interface, das dynamisch erweitert werden soll
- Konkrete Komponente: Kann dynamisch erweitert werden
- Dekorierer: Hält Referenz auf Komponente, Implementiert Interface der Komp.
- Konkreter Dekorierer: Fügt weitere Zuständigkeit zur Komponente hinzu

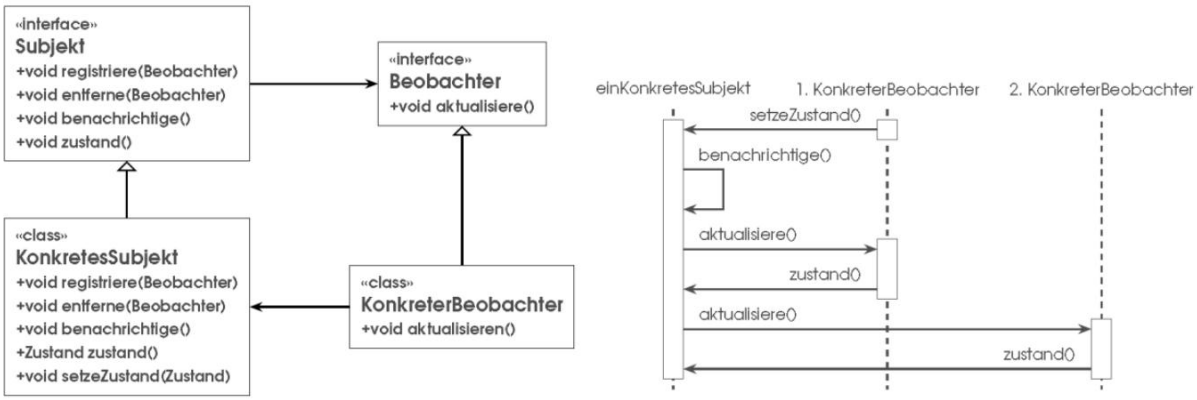


- Auswirkungen
  - + Flexibler als Ableitungen (dynamisch, beliebige Kombination, mehrfach)
  - + Einfache, zusammengesteckte Klassen
  - + Vermeidet große, konfigurierbare Klassen
    -\– Identität des Dekorierers und der Komponente unterschiedlich
  - \– Viele kleine Objekte erschweren das Debuggen bzw. Lernen des System

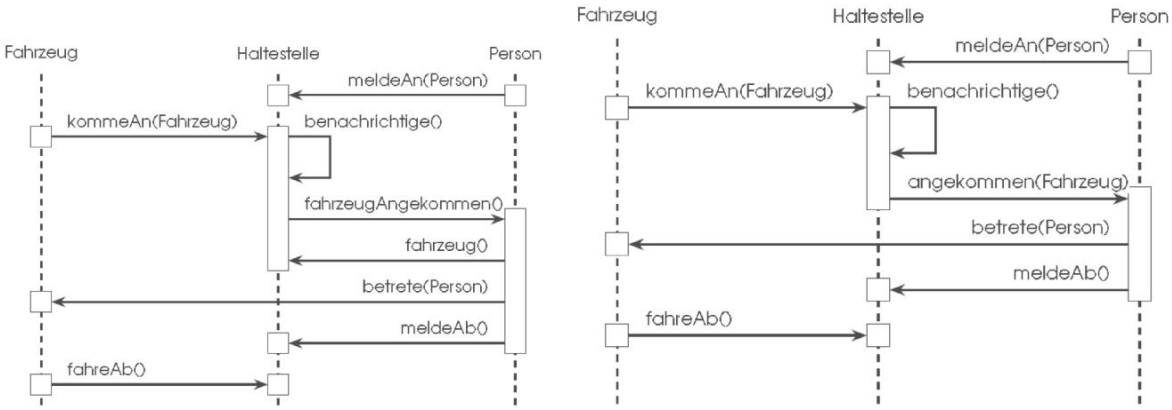
Beobachter

- Definiere 1-zu-viele Beziehungen zwischen Objekten
- Benachrichtige/aktualisiere alle abhängigen Obj., wenn ein Obj. den Zustand ändert
- Langlebig
- Motivation
  - **Sicherstellung bzw. Erhaltung der Konsistenz in modularen Systemen**
  - **Lose Kopplung**
  - **Sofortige Benachrichtigung bei Änderung des Zustands**
- Anwendung

    - Änderung von Obj. zieht andere Änderungen von anderen Obj. unbekannter Zahl nach sich
    - Obj. soll andere benachrichtigen, unabhängig vom Typ
    - Eine Abstraktion hat mehrere Aspekte, die von einem anderen Aspekt derselben Abstraktion abhängen
- Akteure
    - Subjekt: Kennt beliebig viele Beobachter, Interface zur Registrierung und Abmeldung von Beobachtern, Abruf von aktuellem Zustand
    - Konkretes Subjekt: Speichert für Beobachter interessanten Zustand, benachrichtigt Beobachter über Zustandsänderung
    - Beobachter: Schnittstelle zur Benachrichtung bzw. Aktualisierung der Objekte
    - Konkreter Beobachter: Referenz auf konkretes Subjekt, speichert Zustand, der konsistent mit Subjekt sein soll, implementiert Beobachter zur Aktualisierung des Zustands



- Auswirkungen
    - + Nur abstrakte Kopplung zw. Subjekt u. Beobachter über Interface
    - + Automatische Broad-/Multicast-Kommunikation an interessierte Objekte
    - \– Unerwartete Aktualisierung
- **Arten**
    - Push-Modell: Subjekt benachrichtigt Beobachter inklusive Information
    - Pull-Modell: Subjekt benachrichtigt Beobachter exklusive Information, Beobachter muss sie selbst holen



Domain Driven Design

Einführung

Was ist Design ?

- SW beschreibt Ausschnitt aus Realität (Anwendungs-/Problemdomäne)
- Design beschreibt wie ein Modell die realen Gegebenheiten der Problem-Domäne abstrahiert
- Abbildung von Realität auf Modell
- Summe aller Entscheidungen, die Einfluss haben, wie ein Problem als SW-Lösung modelliert wird

Warum braucht man Design?

- SW-Entwicklung ist komplex
- Neben Problemdomäne viele Nebeneinflüsse
- Auswirkungen von Komplexität begrenzbar, kontrollierbar, nicht unnötig verkomplizieren
- Design mach Komplexität beherrschbar

Software-Komplexität

- Grad zu welchem das Design / eine Implementierung schwer zu verstehen ist
- Komplexität der Problemdomäne ist gegeben (essential complexity)
  - Anforderungen
- Unfallkomplexität (accidential comp.): notwendiges Übel, möglichst verhindern
  - Legacy-Systeme, UI, Framework, Persistenz, …
- Mit Lebensdauer von SW wird mehr Code gelesen als geschrieben

Big Ball of Mud

- Code, der etwas nützliches macht, aber ohne Erklärung
- Kein erkennbares Design

-> Code schwer wartbar, erweiterbar

Domain Driven Design

Herangehensweise an die Modellierung von SW, die sich auf Problemdomäne konzentriert

Grundsätze

- Designentscheidungen von Fachlichkeit/Fachlogik der Prob-Domäne getrieben, nicht von technischen Details
- Entwicklung einer Domänensprache (Vokabular)
- Relevante, fachliche Zusammenhänge der Prob-Domäne in Domänenmodell erfassen

Strategie und Taktik

- Strategisch: Verständnis der Domäne, Analysieren, aufdecken, abgrenzen, dokumentieren und begreifen der Fachlichkeit
- Taktisch: Implementierung der Fachlichkeit in Code

Warum ist es wichtig, dass sich die Sprache der Domäne in der Software befindet?

Strategisches Domain Driven Design

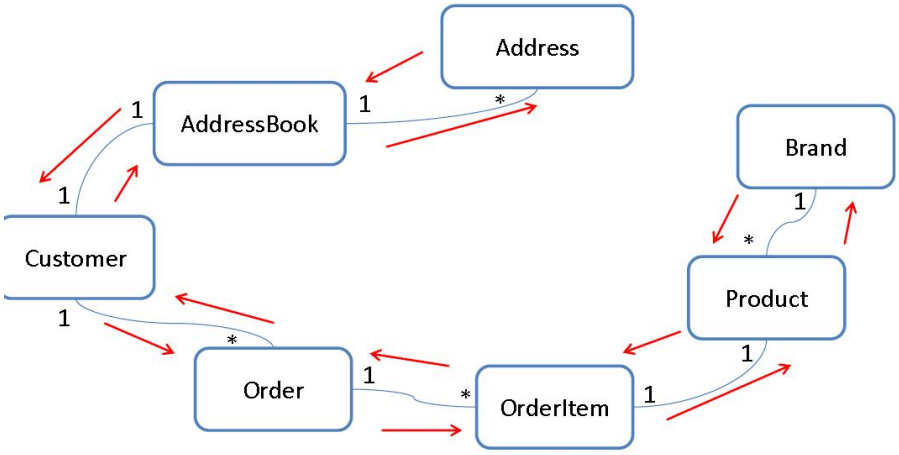
Domäne

- Abgrenzbares Problemfeld oder best. Einsatzbereich für den Einsatz von SW
- Was soll mit SW gelöst werden?

-> Raumfahrt, Logistik, Fertigung, …

Domänenmodell

- Abstraktion, die best. Teile der Domäne beschreibt
- Erlaubt das Lösen von Problemen innerhalb der Domäne
- Wie soll ein Problem mit SW gelöst werden?



Ubiquitous Language

- Domänenmodell erfordert Verständnis
- Weg zum Verständnis der Domäne über Sprache
- Gemeinsame Sprache zw. Domänenexperten und Entwicklern „Ubiquitous Language“
- Technische Sprache der Entwickler --> Fachjargon der Domänenexperten, eignen sich nicht
- Kluft im Verständnis -> pflanzt sich in Code fort
- Ziel der UL: Kluft schließen mit gemeinsamer Sprache (Wichtige Konzepte, Zusammenhänge, Mehrdeutigkeiten / Unklarheiten beseitigen)

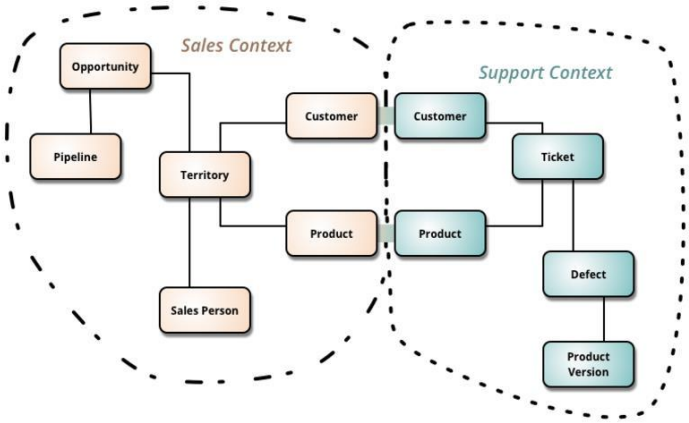
Wie kommt man zur UL?

- Kollaboration zw. Entwicklern / Domänenexperten
- Iterativer Prozess, mit zuhören und nachfragen, eigenes Wörterbuch nur ggf.

Aufteilung der Domäne

Betrachtete Problemdomäne möglichst klein, aufgrund von Komplexität

- Kerndomäne: Kerngeschäft
- Unterstützende Domäne: Unterstützt Kerngeschäft
- Generische Domäne: Nicht Kerngeschäft, z.B. Rechnungen versenden (-> ggf. Dritt-SW)
- Kontext
- Kontextgrenzen: Wo werden Kontexte aufgebrochen? Und wo sind Zusammenhänge? Unabhängig.

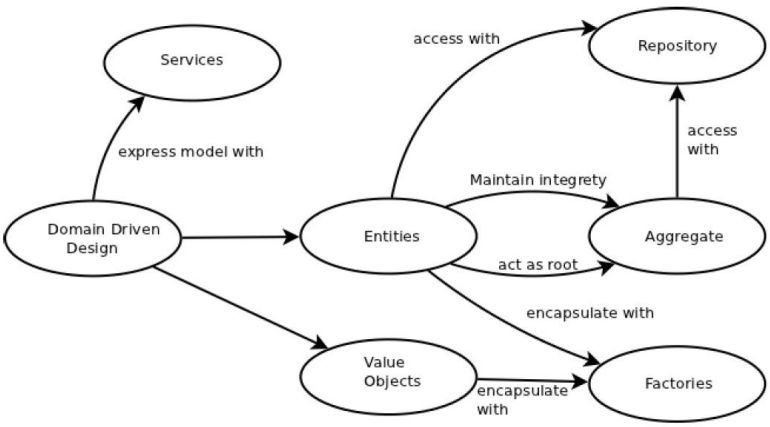


Taktisches Domain Driven Design

Unterstützt bei Entwurf von Modellen, die Komplexität beherrschbar machen, durch Katalog von Entwurfsmustern

Übersicht

- Entities, Value Objects, Domain Services, Aggregates
  - Kern des Modells, Großteil der Geschäftslogik, Forcieren die in der Domäne geltenden Invarianten und machen diese sichtbar
- Repositories, Factories
  - Kapseln der Logik fürs Persistieren/Erzeugen von Entities, Value Objects und Aggregates, Freihalten von Acc.Komplexität
- Modules:
  - Strukturierung/Kapselung verwandter Domänenobj. Innerhalb des Modells, fördern geringe Koppelung & hohe Kohäsion



Value Objects

- Objekte ohne eigene Identität
- Unveränderlich
  - Gültige Konstruktion -> immer gültig (Invarianten einhalten)
  - Frei von Seiteneffekten

- Wird nur durch Werte beschrieben
- 2 VO sind gleich, wenn Werte gleich (equals, hashCode überschreiben)
- Oft Ganzheitliches Konzept
  - Gewicht: Zahl + Einheit
  - Geld: Betrag + Währung
  - Adresse: Straße + PLZ + Stadt
- Beschreiben, begrenzen oder messen Sache näher

Vorteile

- Kapseln Verhalten/Regeln
- Unveränderlich
- Selbst-validierend
- Leicht testbar
- Verbessern Deutlichkeit/Verständlichkeit durch Modellierung von fachlichen Domänenkonzepten

Implementierung

- Klasse: final
- Felder: blank final
- Nach Konstruktion gültig, ansonsten muss Konstruktion fehlschlagen
- Keine Setter
- Rückgabewerte sind unveränderlich oder defensive Kopien

Persistierung

- Eingebettet in Tabelle des Elternobjekts: JPA-Embeddable, jedes Feld eine Spalte
  - Vorteile: Einfach, erlaubt Queries über Elemente des VO
  - Nachteil: Ggf. Denormalisierung, nur 1:1-Beziehungen
- Serialisierung: Objekt in einer Spalte, Converter
  - Vorteile: Komplexe Bez. mög., 1:n-Bez. mög. (Listen, Sets)
  - Nachteil: unlesbar, Queries über VO nicht möglich, aufwändiger
- In eigener Tabelle (als DB-Entity): Entität auf DB-Ebene, mit ID, muss versteckt werden
  - Vorteil: einfach, Normalisierung mög., Queries über VO mög., 1:n-Bez. Mög.
  - Nachteil: Verschleiert Natur von VO durch ID, Gefahr, dass mehrere Entities gleich VO nutzen

Entities

Unterschiede zu VO

- Haben eindeutige ID
- Verschieden, wenn unterschiedliche ID
- Hat Lebenszyklus, verändert sich dabei

Allgemeine Regeln für Entities

- Invarianten forcieren: Konstruktion nur mit gültigen Werten, kein Verändern in ungültigen Zustand
- Möglichst viel Verhalten in VO auslagern
- Die öffentlichen Methoden sollten Verhalten beschreiben (nicht Getter/Setter)
- equals/hashCode: Definition von Gleichheit vom Anwendungsfall abhängig

Strategien für einzigartige Identitäten

Natürliche Schlüssel: Kursname, KFZ-Kennzeichen,

Personalausweisnummer

- Vorteil: aussagekräfig, keine Duplikate wenn global eindeutig
- Nachteil: Fremdbestimmt (ändert sich vielleicht?), ggf. nicht global eindeutig
- Surrogatschlüssel: Selbst generiert

Universally Unique Identifier (UUID)

- Vorteil: jederzeit generierbar, anwendungsübergreifend eindeutig
- Nachteil: nicht sprechend, Performance

Inkrementeller Zähler

- Vorteil: eigenständig, unabhängige ID-Generierung, ID steht sofort fest
- Nachteil: Nicht sprechend, Zähler muss gespeichert werden

String-Format basierend auf Entity-Eigenschaften

- Vorteil: Sprechend, jederzeit generierbar
- Nachteil: hoher Aufwand, falls sich Werte ändern

Surrogatschlüssel vom Perstistence-Provider

- Vorteil: ID eindeutig, kein Aufwand
- Nachteil: nicht sprechend, muss erst durch ORM laufen, abhängig von ORM und DB

Wahl der Strategie

- Selbst verwaltet: Early ID Generation, reduziert Abhängigkeit, erleichtert Komm., wirklich eindeutig?, erleichtert Tests
- Fremdverwaltet: Late ID Generation, erschwert Tests, wenig Eigenverantwortung, funktionierender Standard-Weg

Domain-Service

- Kleiner Helfer innerhalb des Domänenmodells
- Wenn ein best. Verhalten / Regel weder VO noch Entity zugeordnet werden kann
- Beispiele: Berechnung Zahlungsmoral des Kunden (Entities: Kunde, Rechnungen, Kontenbewegungen)

Vertrag

- Domäne kann auf ext. Unterstützung angewiesen sein
- Innerhalb Domäne: Domain Service als Vertrag (Interface)
- Außerhalb Domäne: Dienstleister (Infrastruktur-Schicht) kann Vertrag implementieren und benötige Funktionen bereitstellen

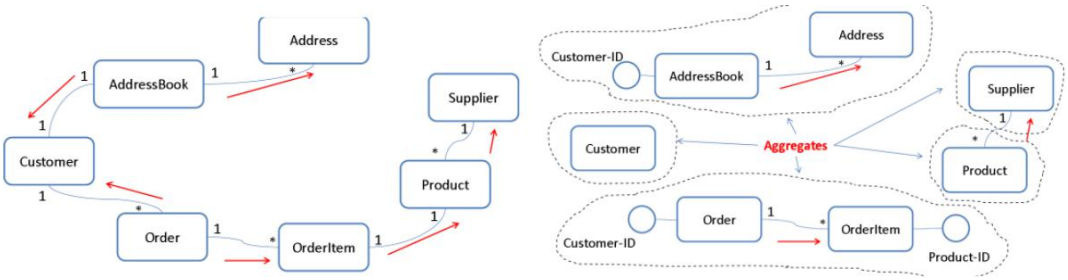
Eigenschaften

- Erfüllt Funktion, die nicht in Entity oder VO modelliert werden kann
- Operiert ausschließlich mit anderen Elementen des Modells
- Verkörpern Konzepte der UL
- Statuslos

Aggregate

- Entstehen von großen Objektgraphen mit bidirektionalen Abhängigkeiten
- Nachteil: Performance, Kollisionen, Verletzen von Invarianten
- Technische Lösung: Locking-Modi, Lazy/Eager Loading
- Domänen-Lösung: Aggregate
- Gruppen Entities und VO zu Einheiten
- Aggregat hat Root Entitiy (Aggregate Root): Über dieses erfolgt Zugriff auf Teile der Aggregate
- Reduzieren Komplexität beim Verwalten von Obj.

- Erleichtern Handhabung von TAs
- Reduzieren die Möglichkeit Invarianten zu verletzen
- Möglichkeiten:
  - Einschränkung der Assoziationsrichtung
  - Ersetzen von Objektreferenzen durch IDs



Ersetzen von Objektreferenten durch IDs

- Entspricht der Abbildung in einer relationalen DB (Foreign Key)
- Nachteile
  - DB über OR-Mapper erzeugt?
    -> keine Fremdschlüssel mehr
  - Weitere Abfragen nötig, um bspw. aufs AddressBook eines Cusomerszuzugreifen
  - Mehr Verantwortung
- Vorteile
  - Schlankere Objektgrapgen
    - Zuständigkeiten klarer getrennt
    - Aggregate bilden bzgl. TAs eine Unit of Work
    - -> Mehr Kontrolle

Aggregate Root

- Alle Zugriffe auf innere Elemente nur über AR
- Referenzen von außen auf innere Elemente nicht erlaubt
- Daher: Zentrale Stelle zur Einhaltung von Invarianten (Bsp: Maximale Bestellpositonen)
- AR sollte nur defensive Kopien ausliefern

Transaktionsgrenzen, Repostitory

- Aggregates werden als Einheit verwaltet (create, read, …)
- Jede Entity gehört zu einem Aggregate
- Repository: Arbeiten mit Aggregates und kann dieses komplett innerhalb einer TA lesen / schreiben

- Aggregates bilden natürliche Transaktionsgrenzen

Zusammenfassung

- Entities und VO zu Aggregates zusammen
- Werden als Einheit betrachtet (create, read, update, …)
- Forcieren/Visualisieren Transaktionsgrenzen
- Forcieren Invarianten
- Persistenz-Frage muss getroffen werden (Nachteile)

Repositories

- Vermitteln zw. Domäne und Datenmodell
- Stellen Domäne Methoden bereit: Lesen etc. von Aggregates aus Persistenzspeicher
- Konkreter Zugriff wird vom Repository verborgen
- Domäne von technischen Details unbeinflusst
- Eigenschaften:
  - Repositories arbeiten ausschließlich mit Aggregates (1:1-Bez)
  - Ähnelt Domain-Service (definiert Vertrag, der in technischer Schicht impl. wird)
  - Aussagekräftige Schnittstelle
  - Ggf. Erzeugung von IDs
  - Kann Prüffelder setzen (LastUpdatedAt)
  - Kann allg. Infos bieten: Zusammenfassung
- Implementierung in technischer Schicht, nicht in Domänenmodell

Factories

- Ggf. Logik für Erzeugen komplex -> Factory
- Erzeugen von Objekten
- Allgemein: Irgendein Objekt/Methode zur Erzeugung anderer Obj. als Kontruktorersatz
- Speziell:
  - Factory Method
  - Abstract Factory

Modules

- Zur Strukturierung/Kapselung verwandter Komponenten
- Abbildung über Namespaces, Packages
- Gruppierung nach fachlichen, nicht technischen, Gesichtspunkten

- Ziel: Zusammengehörende Komponenten gruppieren (hohe Kohäsion, geringe Kopplung zw. Modulen)
- Vorteile: Sichtbarkeit, Abhängigkeiten