Advanced Software Engineering

(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

[Folie 11?]

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 Ausführung

- Einfach ausführbar (1 Knopf), automatisch (kein Input), selbst-überprüfbar (bestanden/fehlgeschlagen)

Thorough (Vollständig)

- Alles Notwendig überprüfen
- Iteratives Vorgehen (bei Fehler ergänzen)
- Fehler 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)

Folie 5?

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

Folie 9?

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 wiederverwenden

- Code Review
- Vor Hinzufügen neuer Funktionalität
- Beim Beheben eines Fehlers

Folie 16?

Wann wird Refactoring schwierig?

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

Auswirkungen auf das Design

Folie 20

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 warden 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

Personen im Umfeld der UI

Folie 5 - 8

Häufiges Vorgehen in der UI-Entwicklung

- Typischer Entwicklungsprozess: Businesslogik -> GUI -> Workflow
- Alle Entwicklungsprozesse durch Programmierer dominiert

- Businesslogik bestimmt GUI und Workflow -> GUI Blooper

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 es beschäftigt ist
    - Lange Operationen ohne Möglichkeit abzubrechen

Responsiveness

1. Responsivness != Performance
2. Ressourcen sind limitiert
3. UI ist ein real-time Interface
4. Verzögerungen sind nicht gleich
5. Aufgaben nicht in der Reihenfolge in der sie erscheinen
6. 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
- Aufgabenspezifisch ohne UI-Begriffe
- Verwendung der Begriffe des Benutzers
- => 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.

Konzeptionelles Objekt	Aktion
Uhrzeit	Anzeigen Einstellen
Alarm	Weckzeit einstellen Musikquelle wählen Wochentage wählen An-/Ausschalten Erstellen Löschen
Musikquelle	Radiosender suchen Speicherort für Musik wählen

Termin werktags

Der Anwender hat montagmorgens einen Termin für ein Vorstellungsgespräch. Dafür will er um 8 Uhr aufstehen.

Zur Vorbereitung auf das Gespräch will er Informationen zu Staus und aktuelle Nachrichten erfahren.

Einkaufen samstags

Der Anwender geht samstagmorgens mit dem Fahrrad auf den Markt einkaufen. Dazu möchte er um 9 Uhr aufstehen.

Aktuelle Nachrichten sind ihm am Wochenende nicht so wichtig. Er hört lieber noch etwas Musik während er gemütlich aufsteht.

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

	Viele Benutzer	**Wenige Benutzer**
Oft verwendet	Gut sichtbar Wenig Klicks	Weniger sichtbar Wenig Klicks
Selten verwendet	Weniger sichtbar Mehrere Klicks	Versteckt Viele Klicks

- 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 Teilnehmerkreises

Häufige 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

- Leitfaden
- Aus Erfahrung und Diskussion
- Existieren in unterschiedlichen Abstraktionen
- Müssen zusammen betrachtet werden
- Müssen nicht erzwungen werden
- Verantwortung aufteilen
- Allgemeiner als Muster

SOLID

- Singe Responsibility
    - Klasse soll nur einen Grund haben sich zu ändern
    - Niedrige Komplexität/Kopplung
    - Jede Klasse nur eine Zuständigkeit
    - Klasse enthält Achsen, auf der sich Anforderungen ändern können (-> nur eine)
    - Beispiel: Bus -> PassangerRoom, Vehicle
- Open Closed
    - Elemente der SW sollten offen für Erweiterungen, geschlossen für Änderungen sein
    - Erweiterung durch Vererbung / Interfaces
    - Bestehender Code wird nicht geändert
    - Beispiel: Loop statt if
- Liskov Substitution
    - Objekte eines abgeleiteten Typs müssen als Ersatz für Instanzen ihres Basistyps funktionieren ohne die Korrektheit des Programms zu ändern
    - Starke Einschränkung der Ableitungsregeln
    - Führt zur Einhaltung von Invarianzen
    - Beispiel: Quadrat, Rechteck, Funktion zur Flächenberechnung
- Interface Segregation
    - Anwender sollten nicht von Funktionen abhängig sein, die sie nicht nutzen
    - Schwere Interfaces/Klasses bündeln viel Funktionalität
    - Interfaces passend zu den Anwendern gestalten
    - Daher oft mehrere Interfaces: Clonable, Serializable, Comparable

    - Beispiel: Arbeiter Interface mit work und eat, Person-Klasse, Roboter-Klasse, Roboter braucht essen eigentlich nicht
- Dependency Inversion
    - High-Level-Module sollten nicht von Low-Level-Module abhängig sein. Beide sollten von Abstraktionen abhängig sein.
    - Details sollten von Abstraktionen abhängig sein.
    - Regeln von High-Level-Modul vorgeben -> Low-Level-Modul implementiert
    - High-Level-Module können wiederverwendet werden
    - Beispiel: Manager abhängig von Worker-Klasse -> Besser Manager abhängig von Worker-Interface, Worker und Superworker können dann das Interface implementieren

TODO: UML-Klassendiagramme

Tell don't ask

- Prozedural
    - Holt sich Informationen, trifft dann Entscheidung
    - Zentralisierte Logik
    - Hohe Kopplung
- OO
    - Sagt Objekten, dass sie etwas tun sollen
    - Verteilte Logik
    - Bessere Kapslung innerhalb der Objekte
- Kommandos stellen besser als Abfragen -> Command Query Seperation (Seiteneffektfrei, definierte Aktionen)

KISS

- Keep it simple, stupid
- Komplexität vermeiden, da Fehler wahrscheinlicher und unverständlich

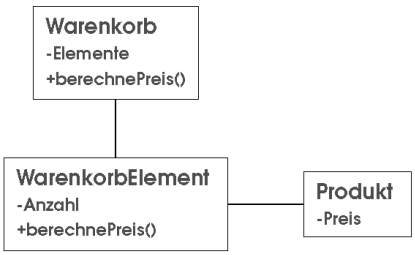
SLAP

- Single Level of Abstraction Principle
- Code in einer Methode ist auf einem Abstraktionsniveau
    - Arbeit, Delegation
    - DB, Businesslogik
- -> Entstehen von Composed Methods
- Fördert Wiedervewendbarkeit
- Sprünge im Abstraktionsniveau schwer zu verstehen

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
- Anwendbar auf alles: DB-Schema, Doku, Tests
- Nur eine Quelle der Wahrheit, alle anderen leiten ab
- Auswirkung der Modifikation haben eine definierte Reichweite
- Arten
    - Imposed Duplication: Auferlegt, unumgänglich
    - Inadvertent Duplication: Versehentlich
    - Impatient Duplication: Ungeduldig, zu faul

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

Entwurfsmuster

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

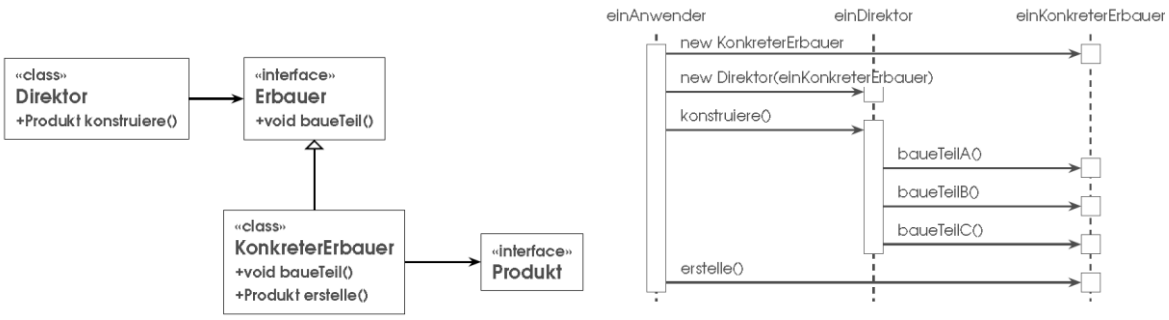
- Zuweisung von Verantwortlichkeiten an Obj.
- Austausch von Verhalten
- Kommunikation zw. Obj.
- Steuerung des Kontrollflusses einer Anwendung zur Laufzeit

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

Folie 10?

Erbauer

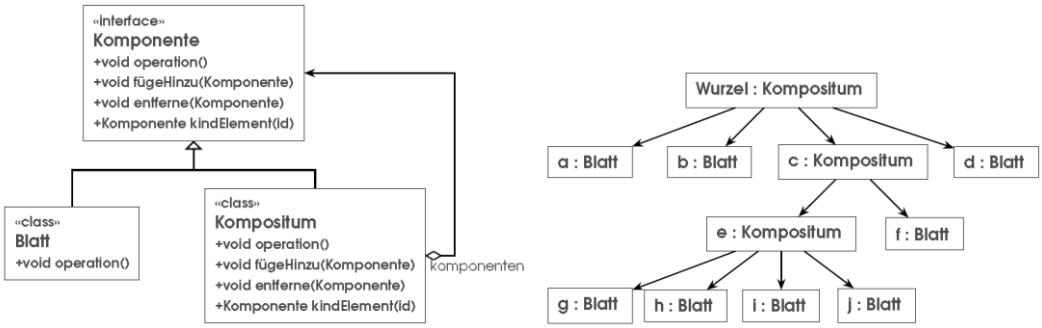
- **Trennung der Erstellung von ihrer Repräsentation von (komplexen) Objekten**
- Gleicher Erstellungsprozess -> ggf. unterschiedliche Repräsentationen
- Wiederverwendung einer komplexen Logik zur Umwandlung von Objekten
- Erzeugungslogik für versch. Formate von Konvertierungslogik trennen
- **Schrittweise 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.
  - Produkt: Komplex erzeugtes Produkt



- Auswirkungen
  - Interne Repräsentation des Prod. kann variieren
  - 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

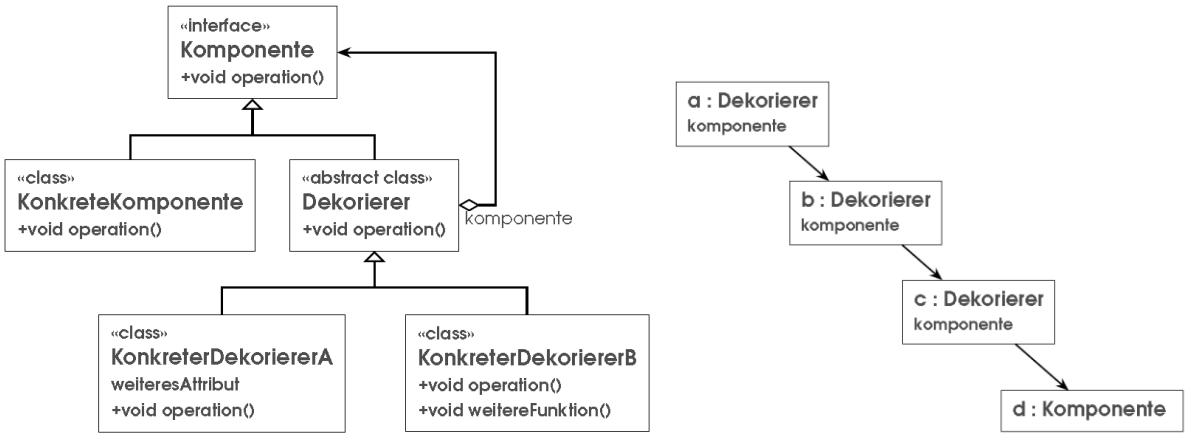


- Auswirkungen
  - + Einfache Elemente können beliebig zusammengebaut werden -> Rekursive Verschachtelung
  - + Vereinfacht Logik beim Anwender
  - + Neue Komponenten können einfach def. werden
  - – Design zu generell
- Transparenz, Typsicherheit
  - Folie 39 – 42?

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 sol
    - Konkrete Komponente: Kann dynamisch erweitert werden
    - Dekorierer: Hält Referenz auf Komponente, Implementiert Interface der Komp.
    - Konkreter Dekorierer: Fügt weitere Zuständigkeit zur Komponente hinzu

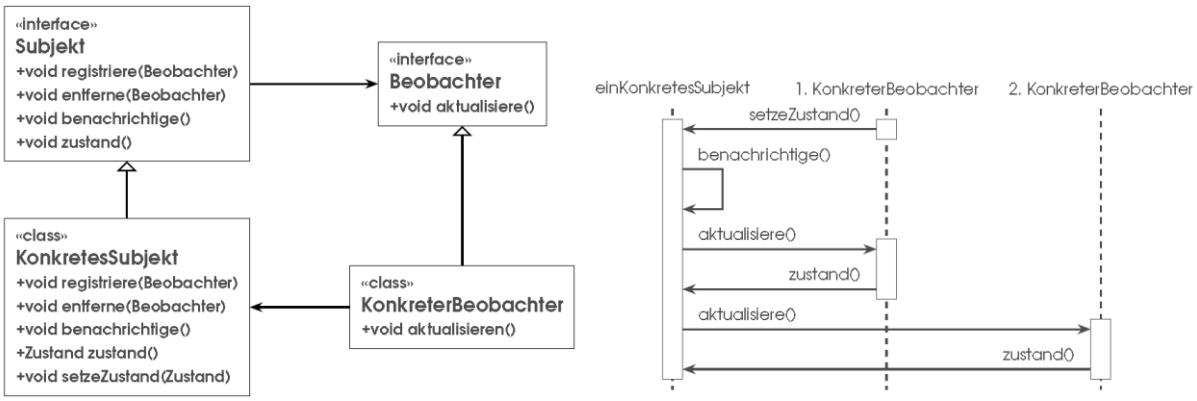


- Auswirkungen
    - + Flexibler als Ableitungen (dynamisch, beliebige Kombination, mehrfach)
    - + Einfache, zusammengesteckte Klassen
    - + Vermeidet große, konfigurierbare Klassen
    - – Identität des Dekorierers und der Komponente unterschiedlich
    - – Viele kleine Objekte erschweren das Debuggen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

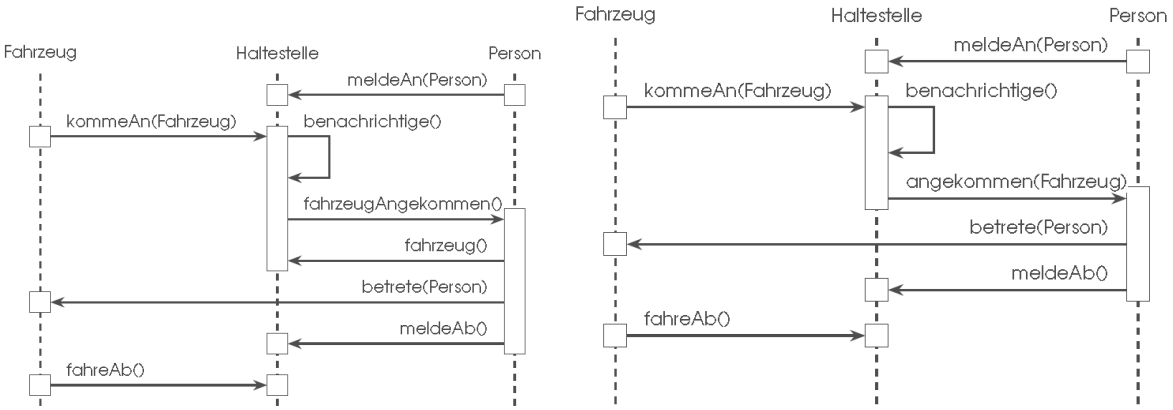
Beobachter

- 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



GUI-Coding

Gestaltungsrichtlinien

- Verwaltungswesen
  - Konkrete Beschreibung einzelner Elemente
  - Einheitliches Erscheinungsbild (Corporate Identity)
  - Inhalt: Papierart, -format, Druckart, Schriftart
- Softwareentwicklung
  - Herstellerspezifisch: MS, Google
  - Platformabhängigkeit

        - Eingabemethode, -genauigkeit, Bildschirmgröße
        - Desktop, Mobile, Web
    - Typische Vorgaben
        - Ist das die richtige Komponente? -> Link oder Button
        - Design-Konzepte (Aktion) -> Print… oder Print
        - Verwendungsarten (Bedienung, Aussehen) -> Normal, Fokussiert, Split Button
        - Richtlinien -> Vorgaben zur Verwendung
        - Größe, Abstände, Farben -> Pixelabstände usw.
        - Beschreibung / Dokumentation ->
    - Inhalt: Fenster, Text, GUI-Komponenten (vgl Blooper)

Testen von Benutzeroberflächen

Motivation und Ziele

- UI ist SW, kann Fehler enthalten -> GUI testen
- Ziel: Funktion von Komponenten u. Zusammenspiel gewährleisten, Test automatisieren
- GUI-Tests: Funktionaler Test von GUI-Komp. (Komponenten-, Integrations-, Akzeptanztest)
- Vorteil: Testen während Buildvorgang, einmal Aufwand für kont. Tests, Test für unterschiedliche Plattformen, Monkey Test vergleichbar mit normalen Nutzer, Screenshots
- Nachteil: Ergebnis genau spezifieren, einmalige Überprüfung durch Menschen schneller
- Probleme: Laufzeit, Asynchron, Grafischer Desktop, Eingabe während Ausführung blockiert

Manuelles Testen

- Reale Person testet Programm von Hand
- Protokoll mit Abfolge von Aktionen
- Fehler/Erfolg wird protokolliert
- Vorteile: Erkennung sinnvoller Abweichung, Plausibilitätsprüfung, Gutes Aussehen prüfbar, schnell bei einmaliger Ausführung
- Nachteil: Zeitaufwändig, teuer, keine Wahrnehmung von Details, erfüllt Anforderung automatisch nicht

Probleme von automatischen Tests

- Suchen/Finden/Interkation der Elemente (-> ggf. ID)
- Überprüfen/Feststellen von Ereignissen
- Protokollierung von Fehlern

Vorgehen

- -> Robot: Zentrale Komponente jedes GUI-Testtools (Benutzeraktionen, Maus-Tastatursteuerung, Screenshots)
- -> Überprüfung: Vergleich Screenshots, Attribute der GUI-Komp., Direkter Zugriff auf Businessmodell
- -> Feststellen von Änderungen: Ereignisse (Events), Polling
- -> Protokollierung: Wie Unit-Tests + Screenshot

Record and Replay

- Aufzeichung -> Aktionen ausführen -> Tests / Überprüfungen definieren
- Erzeugter Test: Testskript, Grafischer Testablauf
- Bestandteile
    - Schrittweise Anleitung
    - Aktion: Pixelkoordinate, Komponente

    - Manuelle eingefügte Überprüfung
    - Ausgabe im Fehlerfall
- Testausführung: Ausführung der gespeicherten Schritte, Aktionen/Fehler protokollieren
- Fehlerfall: Informationen sammeln
- Vorteile: einfach, schnell, wenig Programmieren
- Nachteil: unübersichtlich, manuelle Wiederverwendung, Redundanz, Änderung schwierig

Skriptbasiertes Testen

- Programmierer erstellt Skript
- Vorteil: wartbar, zsmfassen versch. Aktionen, Wiederverwendung, gleiche Programmiersprache
- Nachteil: GUI bei Erstellung nicht sichtbar

Automatisiertes Testen

- Matrix Tests
    - Einsatz: Großer Datenbereich für Eingabe, Kombination versch. Eingaben
    - Ablauf: Auswahl der Parameter, Definition Parawerte -> Tabelle

Addieren	0	1	2	Große Zahl
0	0	1	2	Große Zahl
1		2	3	Große Zahl + 1
2			4	Überlauf
Große Zahl				Überlauf

    -
    - Vorteil: Viele Kombinationen, Reuktion der Redundanz
    - Nachteil: Exponentielle Laufzeit
- Monkey Tests
    - Kontrollierter Zufallsgenerator für Aktionen
    - Suche nach Fehlern
    - Aktionsfolge speichern
    - Bei gefundenem Fehler: Aktionsfolge als dauerhaften Test
- KI
    - Zustände, Aktionen -> Graphensuche für Start- und Endzustand
    - Erzeugung von Testfällen (Graphensuche, EA)
    - Vorteile: Generierung vieler Testfälle, viele Wege zum Ziel testbar
    - Nachteil: Komplexe Definitionen u. Suche

Domain Driven Design

Einführung

Was ist Design?

- SW beschreibt Ausschnitt aus Realität (Anwendungs-/Problemdomäne)
- Design beschreibt wie ein Modell die realen Gegebenheiten der Prob-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?

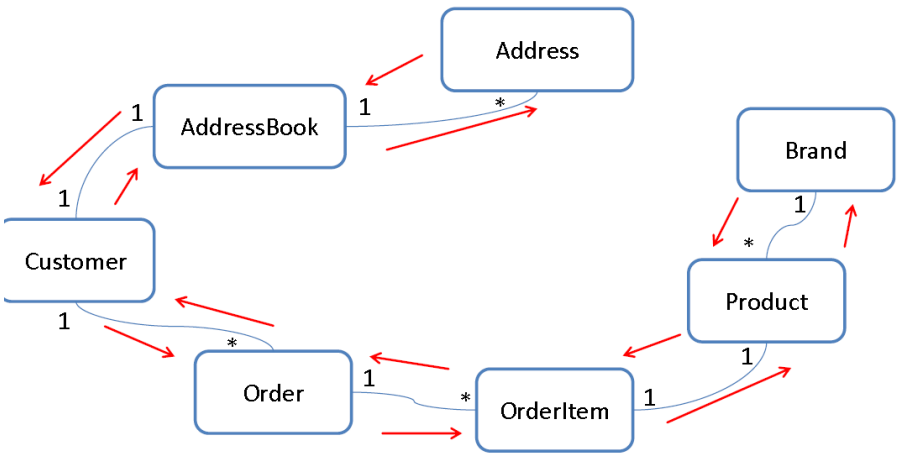
Strategisches Domain Driven Design

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)

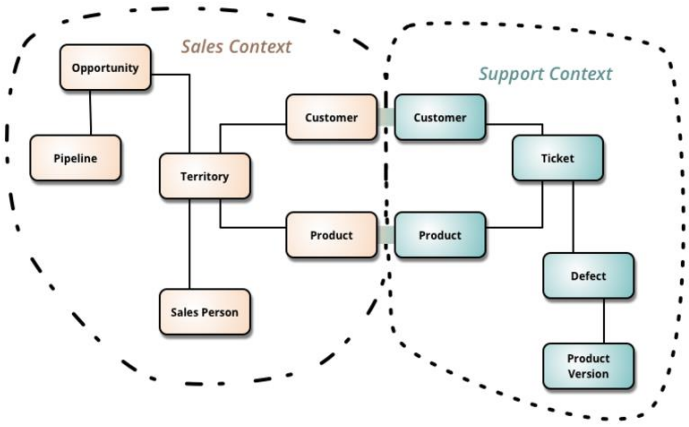
Wie kommt man zur UL?

- 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.

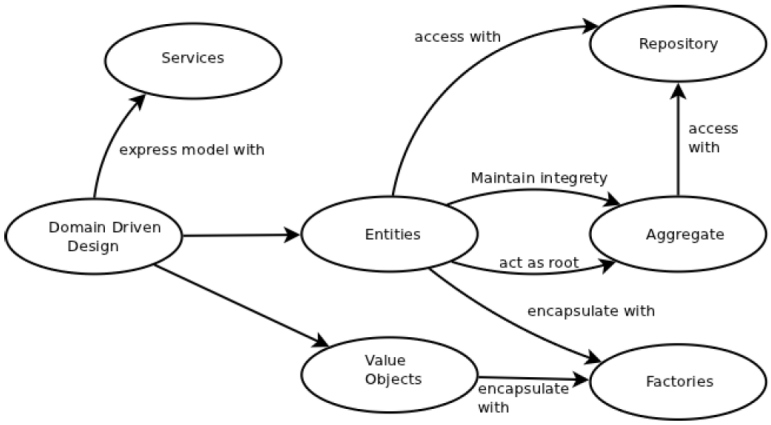


Taktisches Domain Driven Design

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

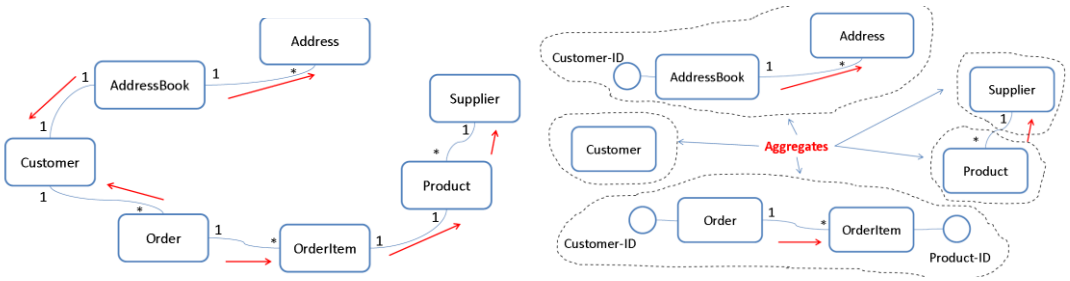
- Domäne kann auf ext. Unterstützung angewiesen sein
- Innerhalb Domäne: Domain Service als Vertrag (Interface)
- Außerhalb Domäne: Dienstleister (Infrastruktur-Schicht) kann Vertrag implementieren und benötige Funktionen bereitstellen

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 Cusomers zuzugreifen
    - -> 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

Onion-Architecture

Klassisch

- Presentation – Business – Infrastructure
- Schicht darf nur mit unterliegenden Schichten kommunizieren
- Strikt: nur nächstniedrigere, Offen: Beliebige Schicht
- Vorteile
    - Beherrschung der Komplexität durch technische Trennung der Anwendung
    - Geringe Kopplung je Schicht, hohe Kohäsion je Schicht

- Einzelne Schichten leichter / unabhängiger änderbar

Dependency Inversion Principle

Motivation

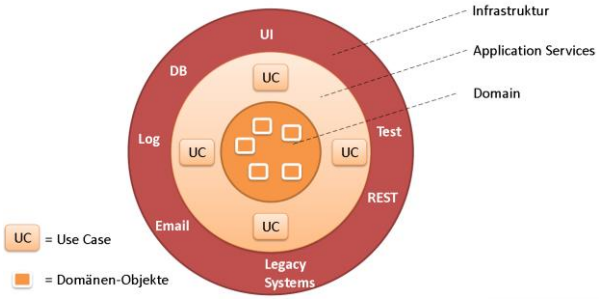
- Repositories ist Teil der Domänenschicht (-> Businessschicht)
- Konkreter Zugriff gehört aber in die Infrastrukturschicht
- Domänenschicht soll frei von technischen Details bleiben
- Implementierung eines Repos nicht in Domänenschicht
- Repo gehört konzeptionell zur Domänenschicht, daher nicht Infrastrukturschicht

Definition

- Domänenschicht gibt durch Interface einen Vertrag (beschreibt, welches Verhalten sie erwartet)
- Konkrete Impl. erfolgt in Infrastrukturschicht
- Domäne nicht abhängig von Details (HibernatePersonRepo), sondern von Abstraktionen (PersonRepo)
- Konkrete Impl. kann dann gewählt werden, bspw. durch Dependency Injection

Umschichtung

- Infrastrukturschicht ist nun abhängig von Domänenschicht
- Verletzt Regel der Kommunikation
- -> Verschieben der Infrastrukturschicht
- Infrastrukurschicht ganz oben
- Darf nun konkrete Verträge für alle anderen Schichten implementieren
- Präsentationsschicht kann als Teil der Infra-Schicht betrachtet werden
- Domänenschicht nun ganz unten, von keinem abhängig -> Kern der Architektur



Onion-Architecture

Hauptmerkmale

- Alle Abhängigkeiten von außen nach innen
- Infrastruktur-Schicht: Technische Details
- Innere Schichten: Frei von technischen Details, UI-Änderungen egal

Domain Layer

- Kernobj., Regeln der Fachlogik (Domänenmodell mit Entities etc.)
- Definiert Verträge (Abstraktionen: Repositories, Logging etc.)

Application Service Layer

- Wenn mehr als Domänenobj. Involviert ist
- -> Implementierung als Application Service
- API für Infrastrukturschicht

- Isolationsschicht zw. Domäne und Infrastruktur
- Infra-Schicht weniger anfällig für Änderung in Domäne

Aufgaben

- Implementierung eines Anwendungsfalls
    - Ein AS -> ein oder mehrere Anwendungsfälle
    - Nutzt Komponenten der Domäne
    - Enthält keine Regeln, weiß nur, welche Domänenobj. in welcher Reihenfolge aufgerufen werden müssen
    - Eher prozedural
    - Keine Create/Update-Methoden -> Konkrete Methoden
- Validierung, Übersetzung / Aufbereitung von Eingaben/Ausgaben
    - Alle Eingaben korrekt / vorhanden?
    - Keine Regeln der Domäne, sondern technische (Datentyp, -format etc.)
    - Übersetzung der Eingaben in Domänenobjekte, ggf. der Ausgabe
- TA-Verwaltung
    - AS als TA, nicht Domänenobjekte
- Reporting
    - Kann Daten aggregieren (Umsätze, Lagerbestände)
    - In Domänenschicht wäre dies teuer -> Datensätze zu Objekte
    - Vertrag erlaubt direkte Zugriff auf Daten
- Security
    - Anwendungsfall durch Zugriffskontrolle sichern

Wie stark soll die Domäne von der Außenschicht abgeschottet sein?

- Objekte dürfen ASL passieren?
- Isolierung bietet Schutz vor ungewollten Veränderungen
- Erhöht Komplexität, zusätzliches Mapping, Indirektion
- Aufwand muss gerechtfertigt sein

Infrastructure Layer

- Technische Details
- Beispiele: UI, Web Services, DB

Kritik

- FW wird isoliert
- Höhere Komplexität (nicht überall benötigt)
- Mehr Aufwand durch Zwischencode und Isolation
- Architektur ist von Testbarkeit getrieben, nicht von Nutzen

DevOps

Was ist DevOps?

Defintion

- Development Operations, Bewegung
- Ziel: Reduzierung der Time-To-Market einer Änderungseinheit - bei gleichzeitiger Gewährleistung hoher Qualität
- Erreichen durch: Konsequente Anwendung von Lean-Prinzipien auf gesamten SW-Wertstrom

Warum DevOps?

- Entwickler sollen Veränderungen schnell umsetzen
- Admins sollen Sicherheit/Stabilität der Systeme gewährtleisten
- -> Zielkonflikt, Silo-Denken (starkes Abteilungsdenken), Stereotypen, Mauer zw. Dev und Ops
- -> Agile Methoden erhöhen Dev-Geschw. -> Mehr Druck auf Ops

Conway's Law

Teams, die nicht gut zusammenarbeiten produzieren Lösungen, die nicht gut zusammenarbeiten

Konsequenzen von Silodenken

- Mehr/Längere Systemausfälle
- Mehr Fehler
- Höheres Risiko
- Angst vor Veränderung
- Längere Time-To-Market, schlechtere Qualität
- Wertstrom stoppt an der Mauer zu Operations
- Viel Zeit geht verloren (Verschwendung) – Features fertig, aber nicht produktiv verfügbar

Folie 18? 26?

DevOps-Prinzipien – CALMS

Culture

Ziel

- Hauptziel von DevOps: Veränderung der Organisationskultur
- Silos zu Kollaboration
- Schuldzuweisung zu gemeinsamer Verantwortung
- Für die erstellten Produkte/Leistungen und Prozesse

Definition von Organisationskultur

- Abstraktes, komplexes Konzept, schwer messbar
- Glaubenssätze, Haltungen, welche ein Kontext für alles in der Organistation bilden
- Hat Einfluss auf Erreichen der Ziele

Kulturmerkmale von DevOps

- Respekt
- Vertrauen
- No-Blame
- Growth Mindset
- Gemeinsame Anreize
- Kollaboration

Blame-Kultur – No-Blame-Kultur

- Nicht Individuen, sondern Situationen führen zu Fehlern
- Suche nach Ursachen statt nach einem Sündenbock
- Blame Kultur: Fehler werden verschleiert (Verschwendung)

5-Why-Methode

- Methode zur Bestimmung von Ursache und Wirkung
- So lange „Warum" fragen bis Ursache klar ist

Fixed Mindset vs. Growth Mindset

- Fixed: Fähigkeiten sind gegeben, Fehler sind Resultat ungenügender Fähigkeiten, Tendenz sich nicht blamieren zu wollen
- Growth: Fähigkeiten können verbessert werden, Fehler sind Gelegenheit zu verbessern, Tendenz nach Rückschlägen weiterzuarbeiten

Gemeinsame Anreize

Team wird belohnt, bei Lieferung von Wert für Kunden, gemeinsame Verbesserung des Flusses von Wert zum Kunden

Kollaboration

- Abhängig vom Unternehmen
- Beispiel: Bereichsübergreifende Teams, Austausch von Experten etc.

Automation

Automatisierung

- Wesentliches Hilfsmittel zur Optimierung des Wertstroms
- Durch: Eliminierung von Verschwendung und Einführung eines def. Arbeitsablaufs
- Wichtig ist, zu wissen, wann es sich lohnt zu automatisieren und wann nicht

Automatisierungsstufen

1. Automatisierte Entwicklungsumgebung (Setup, build, test, check)
2. Continuous Integration
    - Automatisierter Bauvorgang aller Artefakte (deliverables)
    - Automatisierte Tests
    - Automatisierte Code-Analyse
    - **SW kann immer gebaut werden und erfüllt gewisse Qualitätskriterien**
3. Continuous Delivery: CI + Staging
    - Automatisierte Installation/Konfiguration der Zielumgebungen
    - Automatisierte Akzeptanz- und Kapazitätstests
    - Automatisiertes Deployment in Testumgebungen
    - ABER: **Manuelles Auslösen des Deployments in Produktion, SW könnte jederzeit deployed werden**
4. Continuous Deployment : CD + Automatisches Deployment
    - Änderung alle Stages erfolgreich durchlaufen? -> automatisch deployed

DevOps und Automatisierung

- Ziel ist nicht die Automatisierung des Wertstroms
- Bereichsübergreifende Automatisierung zwischen Entwicklung und Produktivbetrieb -> Zusammenarbeit bzgl. Installation und Konfiguration der Zielumgebungen
- Lösung: Infrastructure as Code

Infrastructure as Code

- Infrastruktur wie SW behandeln
- Anweisungen für das Aufsetzen der Umgebungen -> Teil des Quellcodes
- Änderung an Infrastruktur -> Änderung an Quellcode
- Keine Blaupause für ein Unternehmen -> Angepasst an Abläufe, Rechenzentren etc.
- Beispiele: Ansible, Chef

Vorteile von Infrastructure as Code

- Keine Shadow-IT

- Erhöhung der Effizienz
- Infrastruktur wird mit Code kontinuierlich getestet/verwaltet
- Änderungen an Infrastruktur werden versioniert
- Zusammenarbeit von Dev und Ops wird forciert

Wann lohnt sich Automatisierung?

- Wenn im relevanten Maß Zeit gespart wird: Legacy-Systeme einzubinden kostet viel Zeit
- Zeitinvestitionen sind nicht immer vergleichbar (Rollback-Script versus Change Log)
- Zeitersparnis kann auch anderen zugutekommen und multipliziert sich dann

Automatisierungs-Paradoxon

Je höher der Grad der Automatisierung, umso kompetenter der Verwalter

Automatisierungs-Ironie

Grundlagen des Prozesses werden verlernt, Problem: Fehlerfall

Lean

Was ist Lean?

- Philosophie
- Ziel: Kontinuierliche Verbesserung eines Prozesses
- Durch: Eliminierung von Verschwendung, Bedürfnisse der Kunden sind Ausgangspunkt allen Handelns

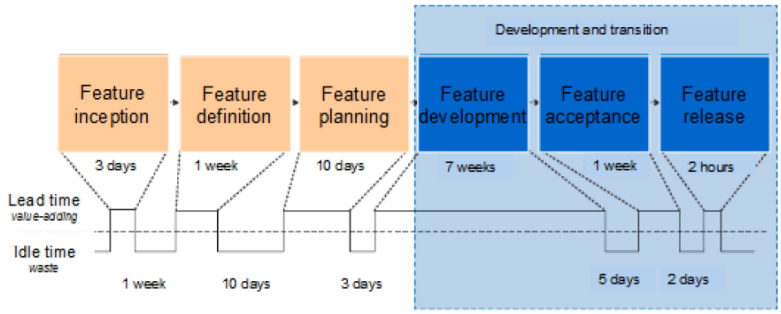
Verschwendungen

- Muda: Menschliche Aktivität, die Ressourcen verbraucht, aber keinen Wert erzeugt
    - Bsp: Materialbewegung, Bestände, Wartezeit, Überproduktion, Korrekturen/Fehler
- Mura: Unausgeglichenheit, Unausgewogenheit, Inkonsistenz
- Muri: Überlastung von Menschen/Maschinen

Lean-Prinzipien

1. Zu produzierenden Wert aus Sicht des Kunden definieren
    - Wert: Leistung, die den Anforderungen des Kunden hinsichtlich Qualität, Verfügbarkeit, Preis entspricht (was der Kunde kauft)
2. Wertstrom identifizieren
    - Alle nicht wertschöpfenden und wertschöpfenden Aktivitäten, die notwendig sind, um ein Produkt/DL herzustellen/anzubieten
    - Ziel des Wertstroms: Kunden zufriedenstellen
    - Jeder Wertstrom umfasst 1 oder mehrere Geschäftsprozesse
    - Unterscheidung nach
        i. Wertschöpfender Tätigkeit
        ii. Nicht wertschöpfende Tätigkeit, die unvermeidbar ist
        iii. Nicht wertschöpfende Tätigkeit, die vermeidbar ist (Verschwendung)
    - Dient insb. der Aufdeckung von Verschwendung
    - Forciert ganzheitliche Betrachtung aller am Wertstrom beteiligten Prozesse
    - Trägt bei zu Siloübergreifende Optimierung des Gesamtprozesses

Beispiel für einen Wertstrom



3. Fluss-Prinzip umsetzen
    - Produkt/Leistung soll zum Kunden fließen, möglichst:
        i. Kontinuierlich
        ii. Mit wenigen Wartezeiten / Umwegen
        iii. Über alle Abteilungen/Stationen hinweg
        iv. In kleinen Änderungseinheiten (z.B. Features)
            1. Geringere Mean Time To Repair
            2. Geringere Time To Market
    - Erreicht durch Beseitigung der identifizierten Verschwendung
4. Pull-Prinzip einführen
    - Nur produzieren, was der Kunde und wann er es will
5. Perfektion anstreben
    - 4 Schritte werden kontinuierlich gemessen und hinterfragt
    - -> Verbesserungspotentiale
    - Ziel: Jede Tätigkeit/jedes Gut trägt zum Wert für den Kunden bei

Metrics

Intention: DevOps messen

- Messen damit kontinuierlich verbessert werden kann
- Klassische Metriken für SW-Entwicklung
- Metriken, die die Leistungsfähigkeit des gesamten Wertstroms aufdecken

Klassische SW-Metriken

- Lines of Code, Zyklomatische Komplexität, Testabdeckung, Stil

Klassische Ops-Metriken

- Uptime, Antwortzeiten, Ressourcen-Auslastung

DevOps-Metriken

- Betreffen gesamten Wertstrom

Durchsatz	**Stabilität**
Häufigkeit von Deployments in Produktion	Mittlere Reparaturzeit
Änderungs-Durchlaufzeit	Fehlerrate in Produktion

- Häufigkeit von Deployments in Produktion
    - Einfach messbar, ggf. Unterscheidung in erfolgreich / nicht erfolgreich
- Änderungs-Durchlaufzeit „Change Lead Time“
    - Zeit von Änderungseinheit von Beginn der Arbeit bis zum Deployment in Produktion
    - Frage: Wann beginnt Arbeit? -> Erfassen der Änderungseinheit
    - = Time To Market
    - Zykluszeit: Ab Beginn der Entwicklung bis Ende der Entwicklung

- Mittlere Reperaturzeit –Mean Time To Repair (MTTR)
    - Zeit zw. Feststellung eines Fehlers in Produktion und dessen Beseitigung
    - Z.B. Durchlaufzeit von Issues „Bug“
- Fehlerrate in Produktion „Change Failure Rate“
    - Zahl der Bugs in Produktion
    - Einfach messbar, ggf. unterscheidbar nach Schwere, Messung von Reaktionszeit

Zusammenhang zwischen Durchsatz und Stabilität

TODO

Share

- Verantwortung, Ziele, Information, Wissen, Code und Tools, …
- Know-How: Verstehen von Ebenen überhalb und unterhalb der eigenen Ebene

Kritik

- Sicherheit/Stabilität manchmal wichtiger als Geschwindigkeit, wird vernachlässigt
- Ggf. Architektur ungeeignet
- Ggf. Anforderung ungeeignet

Fazit

- Abhängig von Organisation, Kultur, Softwareausstattung
- Beginnen mit schneller Auslieferung ohne Qualitätseinbußen
- Herausfinden, wo DevOps Sinn macht: Wo Geschwindigkeit wichtig ist
- Mehr als IT-Methode: Unternehmensweite Transformation
- Kann Produktivität und Markteinführungszeiten von SW verbessern

Ausblick

- DevSec: Sicherheit in den Prozessen verankern
- Bimodale IT: Zwei Arten von IT, traditionell (Tagesgeschäft) und agil (Innovation)

Continuous Delivery

Definition

- Vorgehensweise: soll sicherstellen, dass
    - SW immer in einem auslieferbaren Zustand ist
    - Und automatisiert ausgeliefert (deployed) werden kann
- Durch: Einführung einer hochautomatisierten Deployment-Pipeline (Lean-orientiert)

Antipatterns

- Manuelles Deployment in Produktion
- Entwicklung/Tests nicht in produktivähnlicher Umgebung
- Manuelle Verwaltung der Konfiguration
- Manuelle Tests

Release versus Deploy

- Release: Version einer SW, die für Veröffentlichung außerhalb der Entwicklung geplant ist
- release: Version einer SW wird einer best. Nutzergruppe zur Verfügung gestellt
- Release-Kandidat: Jeder Stand einer SW, der als Release infrage kommt (CD -> jede Version)
- Deploy: Installation und Konfiguration einer Anw. in einer Zielumgebung

Prinzipien

- Wiederholbarer, verlässlicher Auslieferungsprozess
- Möglichst hohe Automatisierung
- Versioniere alles
- Wenn es wehtut -> mach es häufiger (Identifizierung)
- Hohe Qualität
- Fertig heißt deployed (min. auf Testsys.)
- Jeder ist für Auslieferungsprozess verantwortlich (DevOps)
- Kontinuierliche Verbesserung des Prozesses

Deployment-Pipeline

- Code durchläuft mehrere Phasen (Stages)
- Jede Phase ist ein Quality Gate
- Mit jeder Phase: Vertrauen in Funktion/Deploybarkeit steigt, Feedback-Geschwindigkeit sinkt
- Beispiel: Commit Stage -> Akzeptanztest -> Kapazitätstest -> Manuelle Tests -> Release

Bezug zu Lean Manufacturing?

Richtlinien

- Artefakte (Deliverables) werden in Commit-Phase genau einmal gebaut
- Gleiche Deployment-Strategie u. Tools für alle Zielumgebunne
- Nach Deployment: Smoke-Tests
- Alle Tests in produktionsähnlicher Umgebung
- Jede Änderung sofort durch Pipeline
- Fehler? -> Pipeline stoppen
- Jeder Build -> Schlägt fehl oder Release-Candidate

Commit-Phase

Aufgaben

- Kompilieren, Interne Release-Nummer erzeugen
- Automatisierte Tests
- Automatisierte statische Analyse
- Erzeugung aller Artefakte für spätere Phasen (Doku, Testdaten)
    - -> Artefakt-Repository

Artefakt-Repository

- Speichert alle erzeugten Artefakte
- Spätere Phase: über ID auf Artefakt zugreifen
- Verzeichnusstruktur auf Fileserver oft ausreichend
- -> Backup, aufräumen

Akzeptanztest-Phase

Aufgaben

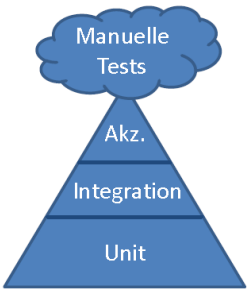
- Zielumgebungen aufsetzen
- Deployment der Artefakte in Zielumgebungen
- Ausführen der Tests: Akzeptanz und Smoke

Smoke-Tests

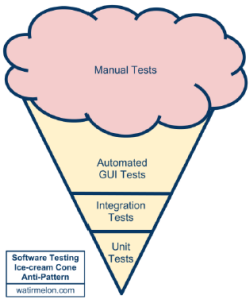
- Testet grundsätzliche Funktion vor Detailprüfung
- Beispiel: Request gg. Anwendung

Verteilung von Tests

- Autom. Akz.-Tests teuer in Erstellung/Unterhaltung
- Wenn, dann: „Happy Path“ testen



Test-Pyramide nach Cohn



Antipattern!!

Weitere Test-Phasen

- Exploratory Testing
- User Acceptance Tests
- Lasttests
- Sicherheitstests

Deployment-Antipatterns

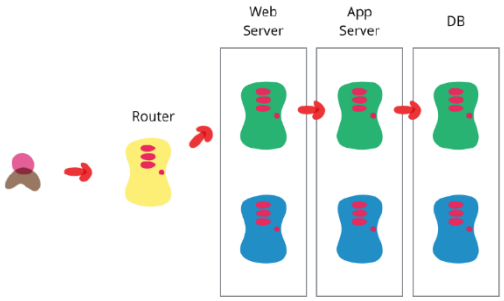
- Doku statt Autom.
- Untersch. Tools für untersch. Umgebungen
- Manuelle Rollbacks
- Manuelle Änderungen an Produktivumgebung

Deployment-Bestpractices

- Automatisierter Rollback-Plan
- Blue-Green-Deployment
- Canary Releases
- Feature Toggles
- Parallel Code Paths

Blue-Green-Deployment

- Zwei identische Produktivumgebungen (doppelte Kosten)
- Eine ist produktiv, andere ist Staging-Umgebung
- Erlaubt schneller Rollback



Canary-Releases

- Beide Version gleichzeitig in Betrieb
- Schritt für Schritt User umleiten
- + Last kann langsam erhöht werden, + schnelles Rollback
- - Höhere Kosten für Infra., - Routing komplex

Features Toggles

- Schalter, um best. Funktionen zu aktivieren
- Möglich, wenn alle im Main-Branch arbeiten
- - unübersichtlich
- -> Features in kleine Einheiten aufteilen

Parallel Code Paths

- Alter/neuer Code parallel ausführen
- Ergebnisse vergleichen -> Abweichung loggen
- Keine Abweichung -> Alten Code entfernen
- + Prüfen von Refactoring

CD-Tools

- Infrastructure as Code-Tools
- Virtualisierung (docker)
- Build-Tools (maven)
- Dependency-Management (maven)
- Dependency-Tools (capistrano)

Schritte zur Deployment-Pipeline

- Wertstrom identifizieren
- Walking Skeleton erstellen
- Build, Deployment autom.
- Unit Tests / Code Analyse autom.
- Akzeptanztests autom.
- Releases autom.

Vorteile

- Weniger Risiko durch häufige Deployments
- Schnelles Feedback
- Mehr Vertrauen / Sicherheit in Funktionsfähigkeit
- Schnelle, wiederholbare Prozesse statt Doku
- Forciert Zusammenarbeit

Nachteile

- Große Umstellung
- Hoher Initialaufwand
- Ggf. hohe Kosten (Akzeptanztests, HW, …)