Techniken der Projektentwicklung

Responsibilities

Franz Kummert, Gerhard Sagerer

Termin 7

Übersicht

1. Ein kleiner Rückblick
2. *GRASP*: General Responsibility Assignment Software Patterns
3. CRC-Karten
4. Ein Beispiel

Was bisher war

Wir haben gelernt,

- wie man Use Cases beschreibt.
- wie man ein Domänenmodell aufstellt.
- wie man Software-Modelle mit Hilfe von UML-Klassendiagrammen beschreibt.

Thema heute

Wie kommen wir zum Softwaremodell?

- *GRASP*: einige Standard-Lösungsansätze
- Low-Tech-Unterstützung durch *CRC*-Karten
- Zentraler Begriff: *Responsibilities* (Verantwortlichkeiten)

Was macht ein gutes Softwaremodell aus?

GRASP

Responsibilities

Was sind Responsibilities?

- *Doing* something:
  - Objekte erzeugen
  - Kontrollfluss steuern
  - ...
- *Knowing* something:
  - Informationen kapseln oder erzeugen
  - Andere Objekte "kennen"
  - ...

Woher kommen Responsibilities?

- Use Cases

Responsibilities

Weg zum Softwaremodell

- Konzeptklassen: keine Responsibilities
- Use Cases:
  $\longrightarrow$ Softwareklassen müssen Responsibilities erfüllen
- Aber: Welche Funktionalität in welche Klasse?

Lösung: *GRASP*

GRASP

Was heißt *GRASP*?

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Auf deutsch: Verhaltensweisen für das Vergeben von Zuständigkeiten

Woraus bestehen *Pattern*?

- Ein Name
- Ein Standardproblem
- Eine Standardlösung

Information Expert

Problem:

- In welche Softwareklasse gehört eine Responsibility?

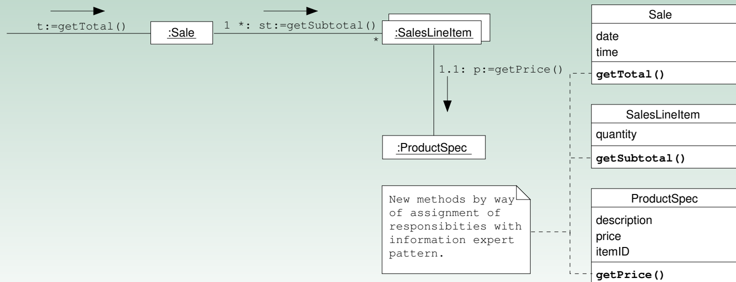
Lösung:

- in die Klasse, die die notwendigen Informationen hat
- sogenannter "Information Expert"
- Daten-zentrierter Ansatz
- falls Softwareklasse bereits vorhanden: alles OK
- falls nicht: Konzeptklasse zu Softwareklasse machen

Information Expert

Beispiel

- NextGen POS: Wir wollen den Gesamtpreis wissen
  $\longrightarrow$ Responsibility: Gesamtpreis ermitteln



Information Expert

Vorteile

- Unterstützt Kapselung → keine unnötigen Abhängigkeiten
- Hoher inhaltlicher Zusammenhang (hohe Kohäsion) → Entwurf leichter verständlich

Probleme

- Reicht (leider) nicht aus
- Beispiel:
  - Responsibility: Speichern eines *Sales* in Datenbank
  - In *Sale* fehl am Platz: soll "einfach ein Verkauf sein"

Creator

Problem:

- Wir benötigen eine neue Instanz einer Klasse A
- Wer erzeugt die Instanz?

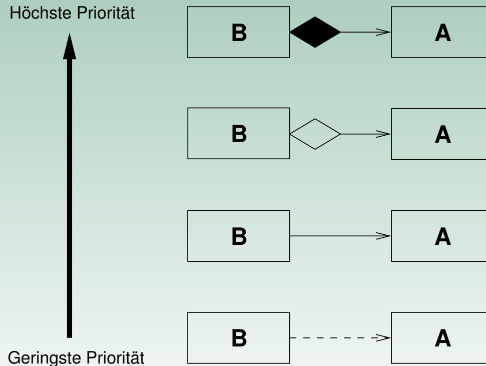
Lösung:

Das Objekt der Klasse B, dass die Klasse A ...

- beinhaltet (Komposition, Aggregation, Assoziation)
- benutzt
- die nötigen Kenntnisse hat

Creator

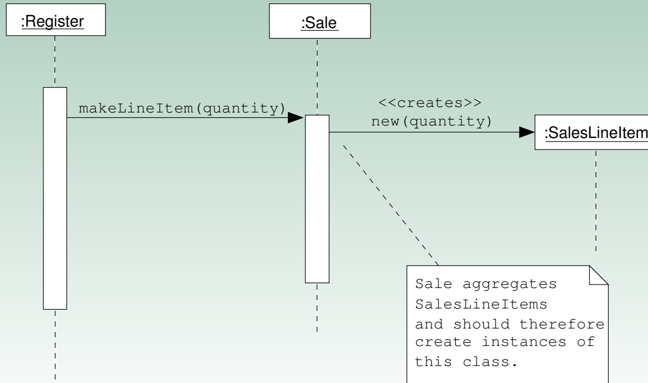
Priorität



Creator

Beispiel

- NextGen POS: Wer erzeugt die *SalesLineItems*?



Creator

Vorteile

- Datenkapselung → siehe *Information Expert*
- Kaum zusätzliche Komplexität
  → Klassen "sehen" sich eh bereits

Alternative

- *Factory*-Pattern: Hilfsklasse (Fabrik), um zu komplexe Instanzierungsvorgänge auszulagern

Low Coupling

Problem:

- Wie erhöhen wir die Wiederverwendbarkeit?
- Wie verhindern wir, dass Änderungen an einzelnen Klasse Änderungen des ganzen Systems nach sich ziehen ("change impact")?

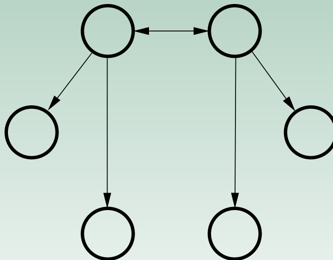
Lösung:

- Responsibilities so zuordnen, dass Koppelung zwischen Klassen gering bleibt

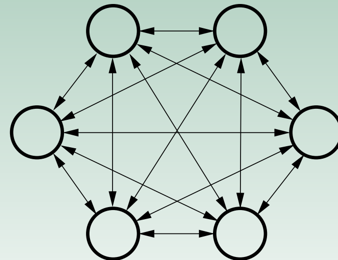
→ Systemkomponenten möglichst unabhängig

Low Coupling

- Klasse soll nur von wenigen Anderen abhängen
- Klasse soll wenig Wissen über Andere benötigen



Gut!

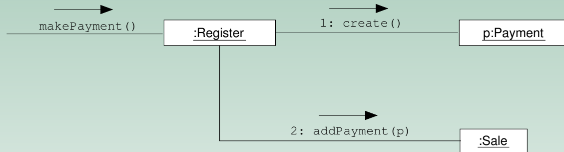


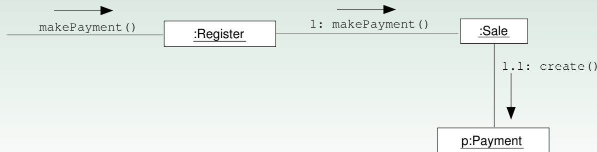
Schlecht!

Low Coupling

Beispiel

- NextGen POS: Wer erzeugt die *Payment*s?





Low Coupling

Vorteile

- kleine Veränderungen berühren nicht das ganze System
- Komponenten einfacher zu verstehen
- Wiederverwendbarkeit gesteigert

Problem:

- niedrigste Kopplung: eine Klasse für alles
- widerspricht OO-Gedanken: Zusammenarbeit von Klassen/Objekten

High Cohesion

Problem:

- Wie halten wir die Komplexität handhabbar?

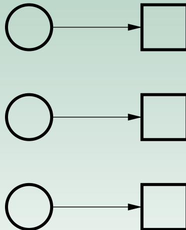
Lösung:

- Responsibilities so zuweisen, dass hohe Kohäsion entsteht
- anders gesagt: Komponenten haben “verwandte” Responsibilities

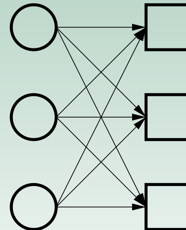
→ hoher funktionaler/inhaltlicher Zusammenhang je Komponente

High Cohesion

- Verantwortlichkeiten eines Elements logisch zusammenhängend
- Element nicht mit Verantwortlichkeiten überfrachtet



Gut!

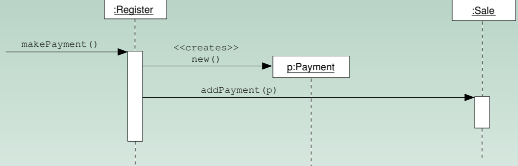


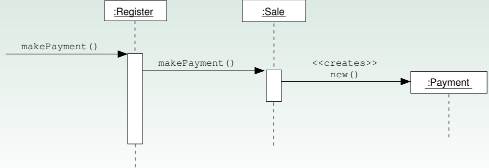
Schlecht!

High Cohesion

Beispiel

- NextGen POS: Wer erzeugt die *Payment*s?





High Cohesion

Vorteile

- Klares und verständliches Design
- Wartbarkeit
- Erweiterbarkeit
- Wiederverwendbarkeit
- bewirkt oft auch *Low Coupling*

Controller

Problem:

- Wer behandelt Systemevents (Eingaben)?

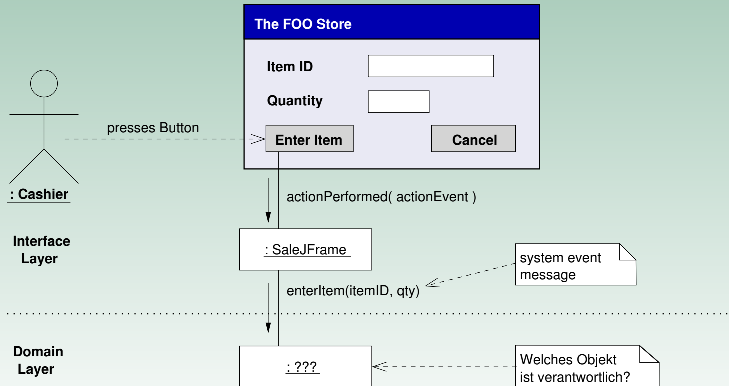
$\longrightarrow$ Interaktion mit Akteuren

Lösung:

- Klassen, die ganze Geräte, Systeme oder Subsysteme repräsentieren: *Facade-Controller*
- Klassen, die bestimmte Use-Cases repräsentieren: *Use-Case-Controller*
- **nicht:** Klassen, die GUI-Elemente repräsentieren

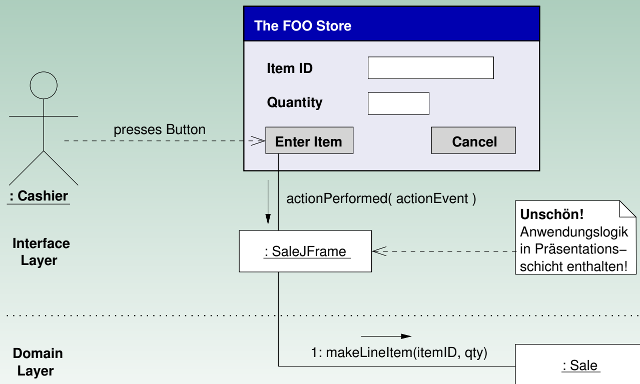
Controller

Beispiel



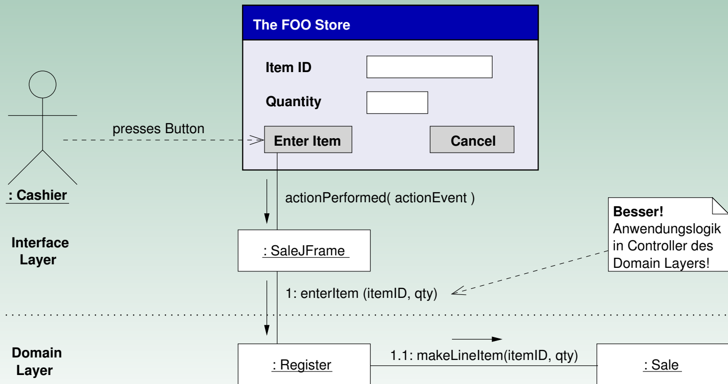
Controller

Beispiel



Controller

Beispiel



Controller

Bemerkungen

- Controller *delegieren* die Arbeit des Systems
- Verrichten selbst kaum Arbeit

Problem: *Bloated Controllers*

- Controller sind zentrale Systembestandteile (Information Experts)
  $\longrightarrow$ werden häufig zu Werkzeugen “für alles”
- niedrige Kohäsion
- mögliche Lösung: mehr Controller definieren

GRASP

Zusammenfassung

- *Information Expert* und *Creator* zum Zuweisen von Responsibilities
- *Low Coupling* und *High Cohesion* zum kontinuierlichen Überprüfen und Bewerten des Designs
- *Controller* verarbeiten Inputs und delegieren an andere Objekte

Hauptquelle: Craig Larman. *Applying UML and Patterns*.

CRC-Karten

Motivation

Ausgangspunkt:

- Domänenmodell
- Use Cases

Ziel:

- Klassenkandidaten bestimmen

$\longrightarrow$ Klassenmodell

Methoden:

- GRASP als Leitschema
- CRC-Karten zum Festhalten von Ergebnissen und als Denkhilfe

Die Idee

Methodik

- Team-orientiertes Vorgehen
- Objektorientiertes Denken: *"No object is an island"*
- Einfache *low-tech* Methode
- Einziges Hilfsmittel: Karteikarten
  $\rightarrow$ Klassen physisch (be-)greifbar

Die Idee

Methodik

- Brainstormingmethode
- Karten visuell Anordnen
  $\longrightarrow$ assoziierte Klassen nahe bei einander
- Durchspielen/Testen von verschiedenen Entwürfen
- Use Cases werden gezielt in Klassen und Responsibilities gegossen

Aufbau einer CRC-Karte

Class - **R**esponsibility - **C**ollaborator

Name der Klasse	Beschreibung durch 1-2 Wörter Beschreibung im Singular
Verantwortlichkeiten Was muss die Klasse wissen? Was muss die Klasse tun?	**Kollaborationen** Mit welchen anderen Klassen arbeitet die Klasse zusammen? ... falls die Klasse Informationen benötigt, die sie selbst nicht hat? ... falls die Klasse Informationen ändert, die sie selbst nicht besitzt?

Eine komplette CRC Modellierung

Ausschnitt aus einer Auftragsverwaltung

Artikel	
Artikelnummer Name Beschreibung Preis Berechne Preis	

Bestellung	
Bestellnummer Bestelldatum Lieferdatum Bestellpositionen Gesamtpreis berechnen Rechnung drucken Stornieren	Bestellposition Kunde

Bestellposition	
Anzahl Artikel Gesamtpreis berechnen	Artikel

Kunde	
Name Telefonnummer Kundennummer Bestellung aufgeben Bestellung stornieren Zahlung durchführen	Bestellung Anschrift

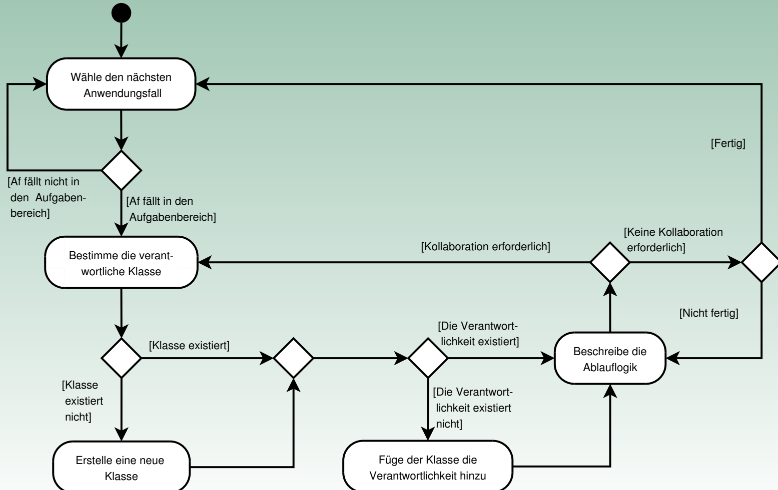
Anschrift	
Straße Ort Land Postleitzahl Etikett drucken	

Durchführung

Der CRC-Algorithmus

1. DO:
   1. Wir wählen einen Use Case und spielen ihn durch
   2. Keine verantwortliche Klasse
      $\longrightarrow$ neue Klasse
   3. Notwendige Responsibilities fehlen
      $\longrightarrow$ neue Responsibilities
   4. Notwendige Kollaborationen fehlen
      $\longrightarrow$ neue Kollaborationen
2. UNTIL:
   - Alle Use Cases abgearbeitet
   - Alle Responsibilities verteilt
   - Alle Kollaborationen notiert

Durchführung

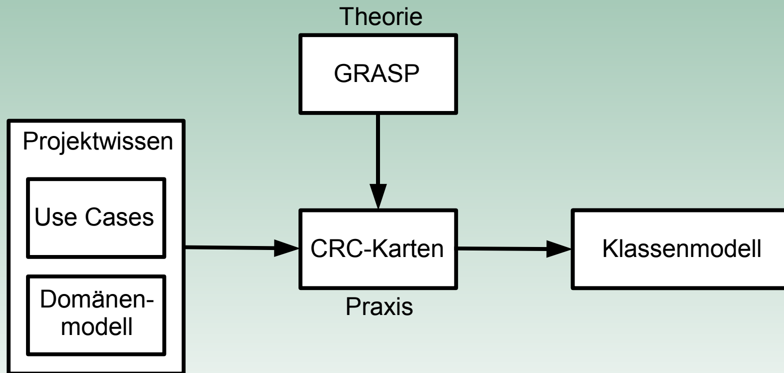


Durchführung

Anwendung von *GRASP*

- Zuweisung der Responsibilities:
  - *Information Expert*
  - *Creator*
  - *Controller*
- Zu viele Responsibilities:
  - Verletzung von *High Cohesion*?
  - Klassen aufteilen
- Zu viele Kollaborationen:
  - Verletzung von *Low Coupling*?
  - Refactoring notwendig

Durchführung



Zusammenfassung

Vorteile

- Wertvolles, gleichzeitig einfaches Instrument
  - Erfordert kaum Einarbeitungszeit
  - Benötigt keine speziellen Werkzeuge
- Einbindung von Benutzern und Domänenexperten möglich
- Durchspielen von verschiedenen Modellen möglich

Ergebnisse

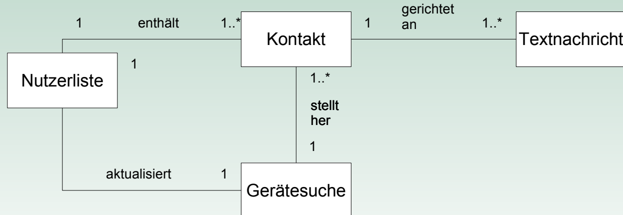
- Validierung der Benutzeranforderungen
- Vollständigere Klassendiagramme
- Notation von Anwendungslogik

Last but not least: Fördert objektorientiertes Denken!

Beispiel

Flirt Factory: Verschicken von Textnachrichten

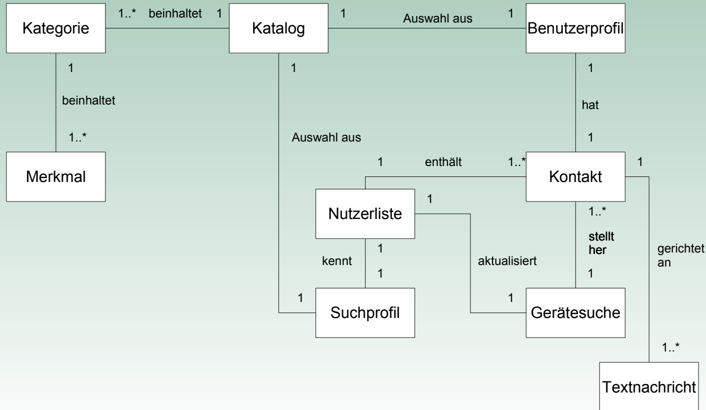
- Aufgabe: Wir wollen eine Textnachricht zwischen zwei Benutzern austauschen
- Responsibilities?
- CRC-Karten?



Hausaufgabe zum nächsten mal

Hausaufgabe

Größerer Ausschnitt aus Flirt Facrory



Hausaufgabe

Use Case: Nutzer in Reichweite lokalisieren

Use Case: NutzerInnen in P2P - Reichweite suchen	
Anwendungsfallname	NutzerInnen in P2P - Reichweite suchen
Hauptakteur	System
Nebenakteure	---
Auslöser	System startet die Suche nach NutzerInnen
Vorbedingung	Dienst aktiv
Erfolgszustand	Suche ist abgeschlossen
Fehlerzustände	---
Hauptszenario	1. System startet Bluetoothsuche 2. NutzerInnen in Reichweite werden gespeichert 3. (Buddies in Umgebung lokalisieren) 4. (Passende ServicenutzerInnen in Umgebung lokalisieren)
Nebenszenario	---

Hausaufgabe

Use Case: Nutzer in Reichweite lokalisieren

Use Case: Passende ServicenutzerInnen in Umgebung lokalisieren	
Anwendungsfallname	Passende ServicenutzerInnen in Umgebung lokalisieren
Hauptakteur	System
Nebenakteure	---
Auslöser	(NutzerInnen in P2P - Reichweite suchen)
Vorbedingung	Dienst aktiv
Erfolgszustand	Passende ServicenutzerInnenliste in Reichweite aktualisiert
Fehlerzustände	---
Hauptszenario	1. Profilübereinstimmungen mit Nutzern außer Buddies werden berechnet 2. Speichern der ServicenutzerInnen ab Übereinstimmungsgrad X
Nebenszenario	---

Hausaufgabe

Zum nächsten mal:

- Responsibilities bestimmen
- CRC-Karten erstellen und testen
- Abgabe bis 12 Uhr am Vortag des nächsten Tutoriums
- Abgabe der CRC-Karten als PDF unter `/vol/tdpe/groupX/session7/teamY.pdf`
- CRC-Karten mitbringen!