Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InIT Institut für angewandte
Informationstechnologie

Bachelor of Science (BSc) in Informatik

Modul Software-Entwicklung 1 (SWEN1)

LE 07 – Use Case Realization Zusammenfassung

SWEN1/PM3 Team:
R. Ferri (feit), D. Liebhart (lieh), K. Bleisch (bles), G. Wyder (wydg)

Ausgabe: HS24

Um was geht es?

- Wie kann ich aus den Analyse- und Design-Artefakten die eigentliche Realisierung der Use-Cases machen?
- Wie wende ich GRASP Patterns korrekt an?
- Wie komme ich schlussendlich zu lauffähigem und korrektem Code?

Lernziele LE 07 – Use Case Realization

- Sie sind in der Lage:
  - Use Cases zu realisieren, was Sie mit UML-Artefakten dokumentieren
  - den Quellcode aus den Design Artefakten abzuleiten

Agenda

1. Einfluss Analyse Artefakte
2. UML und Design to Code
3. Repetition GRASP
4. Vorgehen
5. Fallstudie «NextGenPos»
6. Fallstudie «Monopoly»
7. Wrap-up und Ausblick

Denkpause

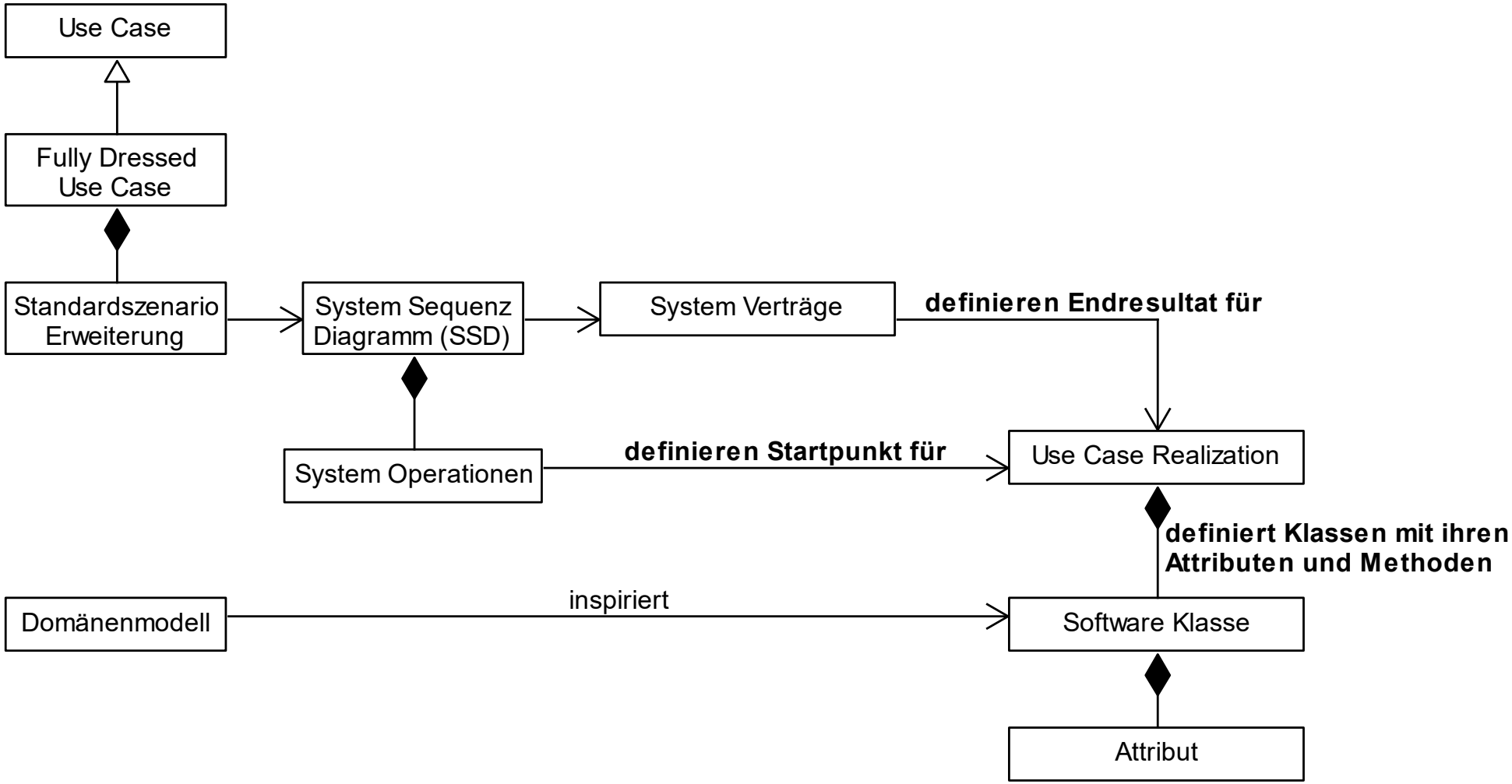
Aufgabe 7.1 (5’)

Wir haben in den LE02 – LE04 verschieden Artefakte kennengelernt, um Anforderungen zu spezifizieren. Dazu gehören:

- Use Cases
- System-Sequenzdiagramm (SSD)
- Operation Contract (Systemverträge)
- Domänenmodell

Repetieren Sie, zuerst ohne nachzuschlagen, den Aufbau und die Varianten dieser Artefakte. Schätzen Sie ab, welche Information daraus für die Realisierung von grossem Nutzen sind.

Analyse Artefakte und Use-Case-Realization



Agenda

1. Einfluss Analyse Artefakte
2. UML und Design to Code
3. Repetition GRASP
4. Vorgehen
5. Fallstudie «NextGenPos»
6. Fallstudie «Monopoly»
7. Wrap-up und Ausblick

Warum UML? (1/2)

- Das zu erreichende Ziel der Use-Case-Realization ist der lauffähige und korrekte Code.
- UML (Klassendiagramm, Interaktionsdiagramme) kann dafür als Zwischenschritt verwendet werden.
  - Empfehlenswert für alle, die noch wenig Erfahrung in der Software Entwicklung haben.
  - Hängt von den Vorgaben der Organisation ab, in der man tätig ist.
- In diesem Modul, und speziell in dieser Lerneinheit, wird UML als Ersatz für eine Programmiersprache aus didaktischen Gründen verwendet.
  - Keine Details der Programmiersprache, die den Lerninhalt «vernebeln».
  - Zusammenarbeit der Klassen ist klarer sichtbar.
  - Kompaktere Darstellung der wesentlichen Aspekte.

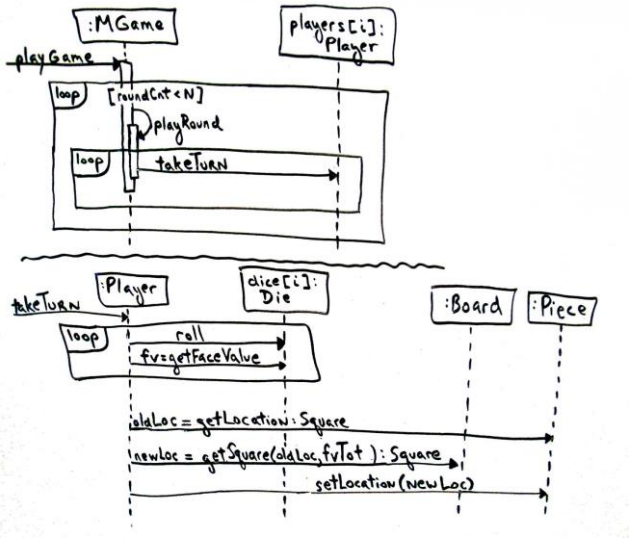
Warum UML? (2/2)

- Für eine nicht beteiligte Person ist UML einfacher zu verstehen als der reine Code.
- In der Praxis, insbesondere bei agilen Methoden, wird auf eine vollständige Dokumentation mit UML verzichtet.
- Es kann aber gut möglich sein, dass die Organisation, für die Sie arbeiten, gewisse Dokumentationsvorgaben stellt. In diesem Fall ist der Einsatz von UML zu prüfen.

Übersicht Design -> Code

Welche Informationen können Sie aus den Design Artefakten für die eigentliche Implementierung ableiten? Und in welchem Detaillierungsgrad?

Design Artefakte



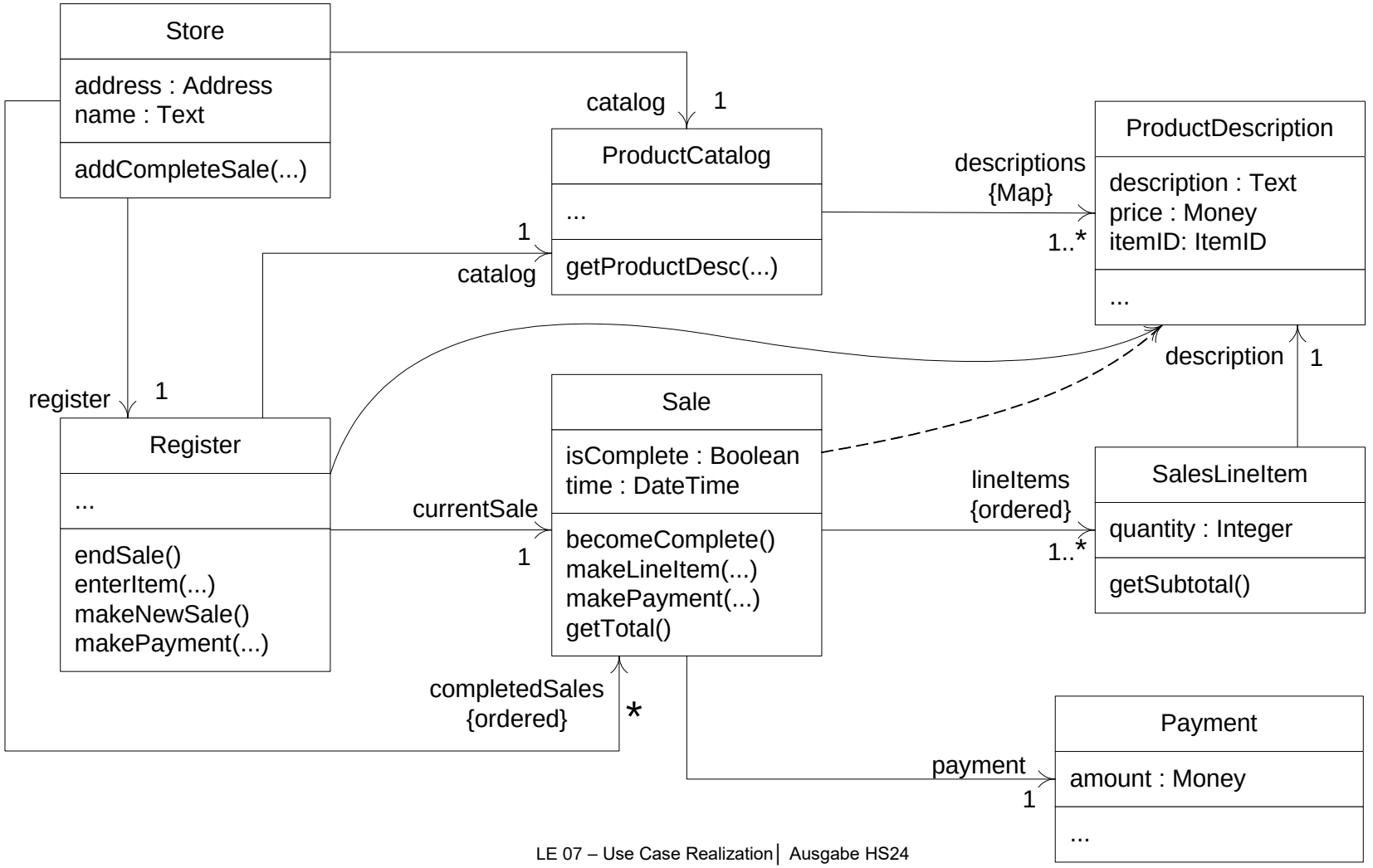
Ausführbarer Code

```
public class MonopolyGame {
   public void playGame(){
       int roundCnt = 0;
       while(roundCnt < 1000){
           PlayRound();
       }
   }

   private void playRound(){
       // ….
   }
}
```

Beispiel Fallstudie: NextGenPos
DCD - Design Class Diagramm

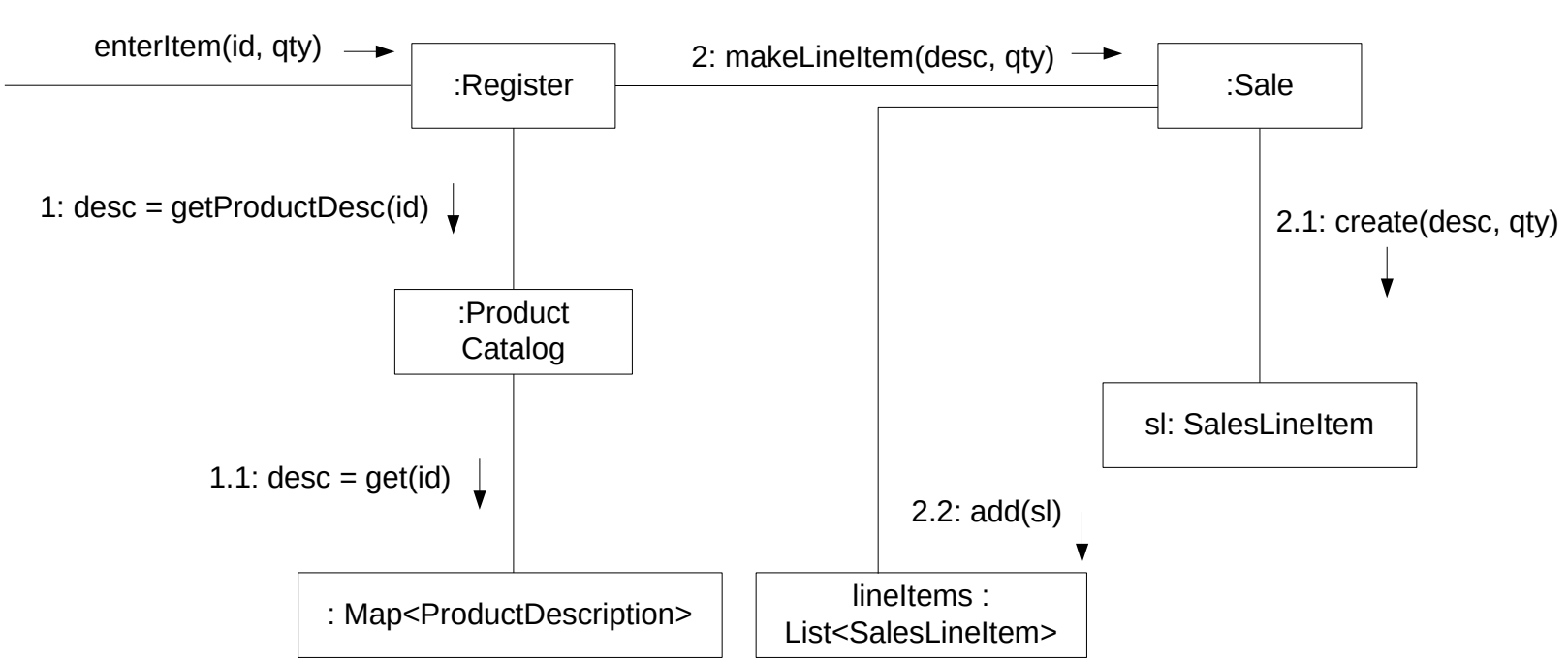
- Klassen
- Attribute
- Methoden
- Assoziation



Beispiel Fallstudie: NextGenPos
Methoden aus Interaktionsdiagrammen

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

- Methoden mit Signaturen




```
Register.enterItem(int itemId, int qty);
// Zwei Ereignisse werden an sichtbare Klassen gesendet
ProductDesription desc = catalog.getProductDescription(itemId);
currentSale.makeLineItem(desc, qty);
```

Agenda

1. Einfluss Analyse Artefakte
2. UML und Design to Code
3. Repetition GRASP
4. Vorgehen
5. Fallstudie «NextGenPos»
6. Fallstudie «Monopoly»
7. Wrap-up und Ausblick

Denkpause

Aufgabe 7.2 (5’)

Wir haben in den LE06 die GRASP Prinzipien kennengelernt für ein gutes, objektorientiertes Design:

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Repetieren Sie, zuerst ohne nachzuschlagen, diese Prinzipien.

GRASP – Die 5 wichtigsten Prinzipien

Für die Use-Case-Realization sind insbesondere die 5 ersten Prinzipien zentral:

- Information Expert
  - Eine Klasse bekommt die Verantwortlichkeit, wofür sie die notwendigen Informationen hat
- Creator
  - 3 Regeln, um die Erzeugung einer Instanz einer Klasse zuzuweisen
- Controller
  - Der Fassaden-Controller übernimmt als erste Instanz vom UI die Ausführung der Systemoperationen
- Low Coupling
  - Bei mehreren Varianten ist die zu bevorzugen, die weniger Kopplungen hat
- High Cohesion
  - Die Verantwortlichkeiten einer Klasse sollten möglichst kohäsiv («fokussiert») sein.

Agenda

1. Einfluss Analyse Artefakte
2. UML und Design to Code
3. Repetition GRASP
4. Vorgehen
5. Fallstudie «NextGenPos»
6. Fallstudie «Monopoly»
7. Wrap-up und Ausblick

Vorbereitung

1. Use Case auswählen, offene Fragen klären, SSD ableiten
2. Systemoperation auswählen
3. Operation Contract (Systemvertrag) für diese Systemoperation erstellen/überlegen/lesen
4. Aktueller Code/Dokumentation des relevanten Teils der Software analysieren.
   1. DCD überprüfen/aktualisieren
   2. Vergleich mit relevantem Teil des Domänenmodells durchführen
   3. Allenfalls bereits jetzt neue Software Klassen erstellen gemäss Vorlage Domänenmodell
5. Falls notwendig, Refactorings durchführen

Vorgehen

1. Controller Klasse bestimmen resp. identifizieren
   - Siehe GRASP Controller Pattern
2. Zu verändernde Klassen festlegen
3. Weg zu diesen Klassen festlegen
   - Allenfalls mit Hilfe von Parametern den richtigen Weg auswählen
   - Allenfalls Klassen, die notwendig sind, neu erstellen
   - Immer Aufruf weiterleiten mit allen noch notwendigen Parametern
   - Verantwortlichkeiten gemäss GRASP Information Expert zuweisen
   - In Varianten denken, Varianten gemäss Low Coupling und High Cohesion bewerten.
4. Veränderungen gemäss Systemvertrag programmieren
5. Review bezüglich High Cohesion und Architekturkonformität

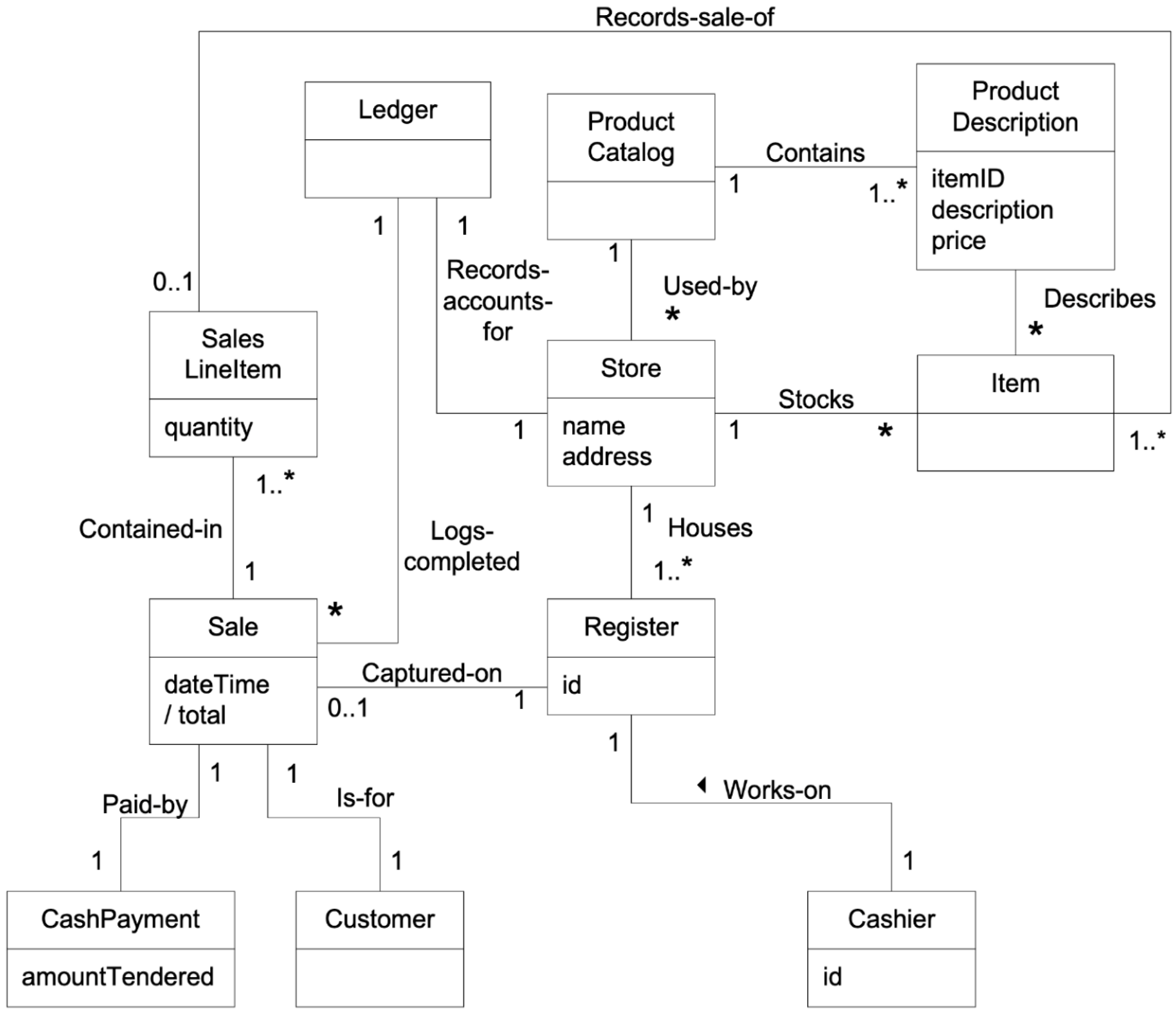
Agenda

1. Einfluss Analyse Artefakte
2. UML und Design to Code
3. Repetition GRASP
4. Vorgehen
5. Fallstudie «NextGenPos»
6. Fallstudie «Monopoly»
7. Wrap-up und Ausblick

Einleitung

- Wir führen nun die Use-Case Realization von UC Process Sale des Projekts NextGenPos durch. Es ist das Fallbeispiel aus Larman[1]
  - makeNewSale()
  - enterItem(idemId, quantity)
  - endSale()
  - getTotal()
  - makePayment()
- UML nochmal kurz erwähnt:
  - Es folgen sehr viele UML-Artefakte und schriftliche Erklärungen.
  - Dies geschieht aus didaktischen Gründen.
  - Die Überlegungen dahinter sollten Sie auf alle Fälle machen.
  - Wieviel UML in der Praxis gezeichnet wird, ist eine Frage, die wir bereits erörtert haben.

Analyse Artefakte: Domänenmodell



Analyse Artefakte: Use Case und SSD

Use Case, Standardszenario

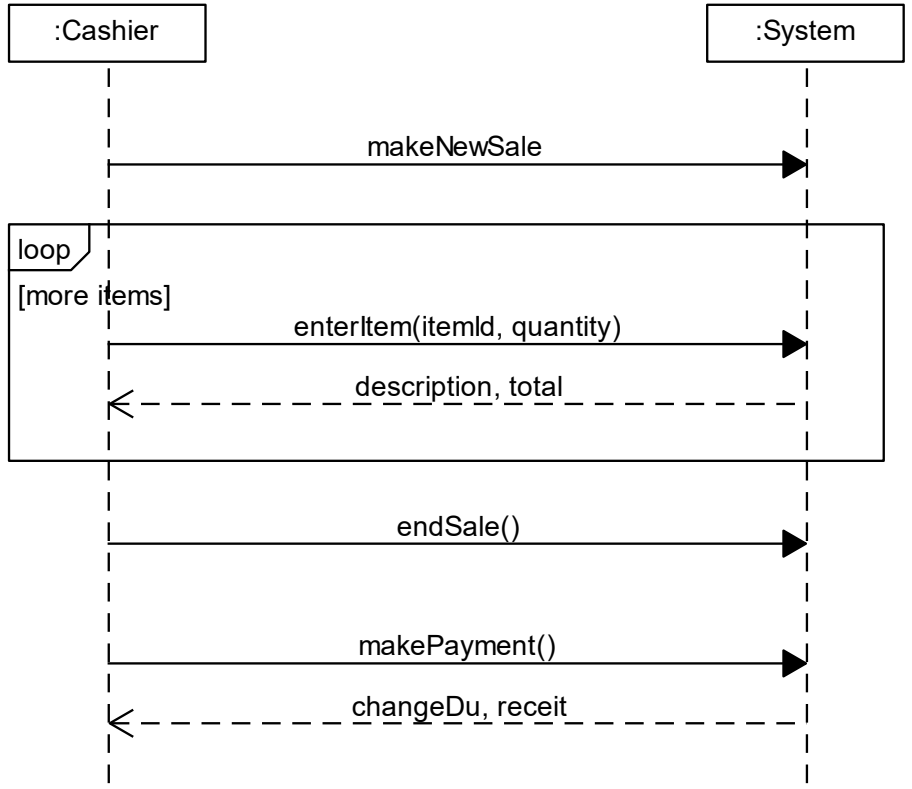
Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.

Cashier repeats steps 3-4 until indicates done.

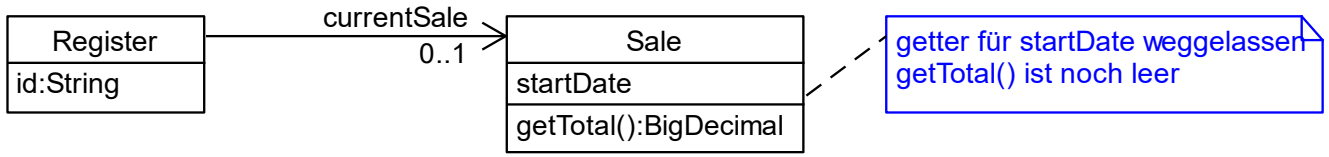
5. System presents total with taxes calculated.
6. Cashier tells Customer the total and asks for payment.
7. Customer pays and System handles payment.

System Sequenz Diagramm



makeNewSale (1)

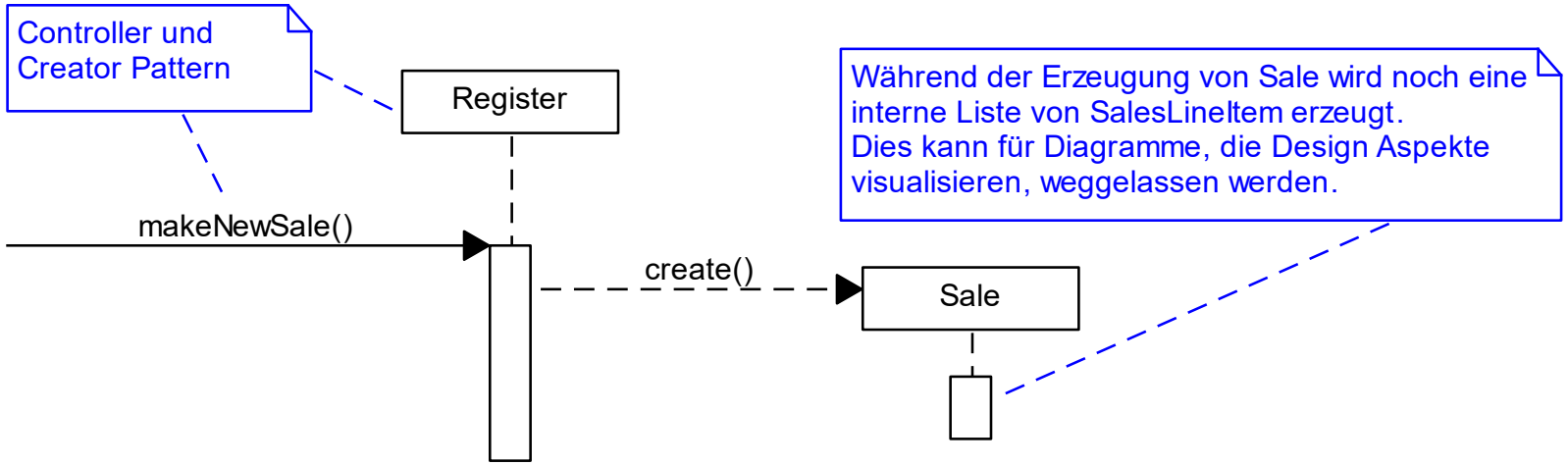
- Vorbereitungsarbeiten
  1. Use Case «Process Sale», fully dressed ausgearbeitet
  2. Systemoperation «makeNewSale()
  3. Operation Contract, Nachbedingungen
     - `Neue Sale`-Instanz `s` ist erstellt
     - S ist die neue aktuelle Sale Instanz von Register
  4. Aktueller Status der Software: Erste Systemoperation überhaupt, noch nichts erstellt
  5. Klassen Register und Sale gemäss DM erstellen



makeNewSale (2)

- Controller Klasse bestimmen: GRASP, Controller Pattern anwenden
  - Für kleine Anwendungen reicht ein Fassaden Controller, der normalerweise das System darstellt, das neu entwickelt wird
  - Register ist das System, daher wird die SW-Klasse die Controller Klasse
- Veränderungen festlegen
  - Neue Instanz von Sale erstellen
  - Diese als neue aktuelle Sale Instanz in Register ablegen
- Weg zum Ziel festlegen: Register ist bereits der Controller, wir brauchen keine Zwischenschritte.
- Sale wird erzeugt: Creator Pattern anwenden. Register ist der Container von Sale, daher soll Register eine neue Instanz von Sale erzeugen.

makeNewSale (3)



Review

- Die Kohäsion von Register als Controller kann heikel sein, wenn dem Controller zu viel Verantwortlichkeit zugeteilt wird. Hier hat er aber Aufgaben, für die es keine andere sinnvolle Domänenklasse gibt.
- Das Schichten-Prinzip wurde eingehalten.

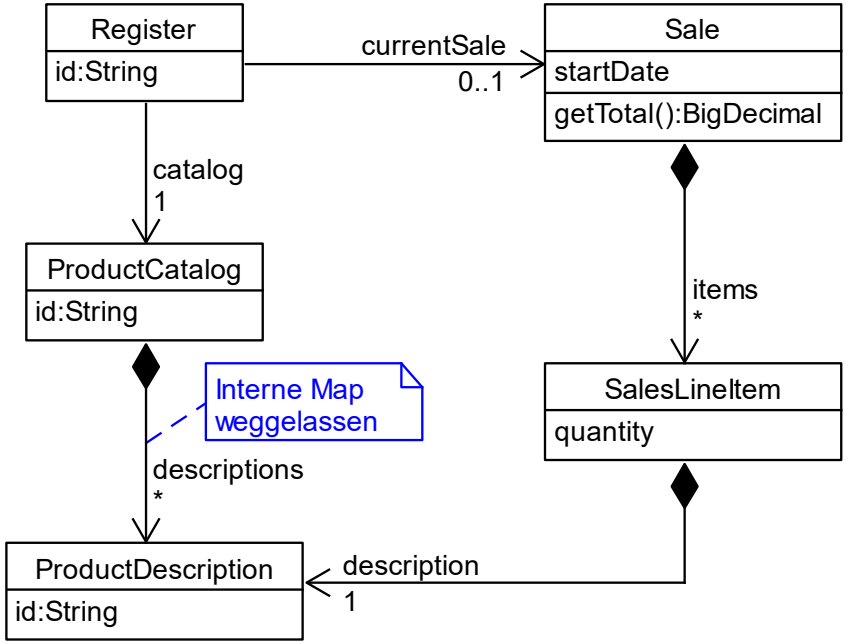
EnterItem (1)

Vorbereitungsarbeiten

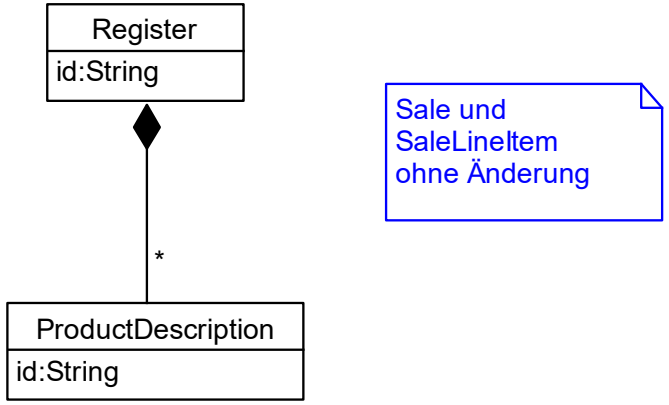
1. Use Case «Process Sale», fully dressed ausgearbeitet
2. Systemoperation «enterItem(itemId, quantity)
3. Operation Contract, Nachbedingungen
   - `SaleLineItem`-Instanz `sli` (ist) erstellt
   - `sli` mit aktueller `Sale`-Instanz verknüpft
   - `sli.quantity` auf `quantity` gesetzt
   - `sli` mit entsprechender `ProductDescription` verknüpft (gemäss `itemID`)
4. Aktueller Status der Software: Siehe vorhergehendes Beispiel
   1. Klassen Register und Sale erstellt mit wenigen Methoden
5. Klasse ProductDescription wird im Operation Contract erwähnt und daher gemäss DM erstellt. Dort sehen wir, dass es noch einen ProductCatalog als Behälter von ProductDescriptions gibt, daher erstellen wir auch diese Klasse.

EnterItem (2)

Klassendiagramm:



Klassendiagramm Variante:

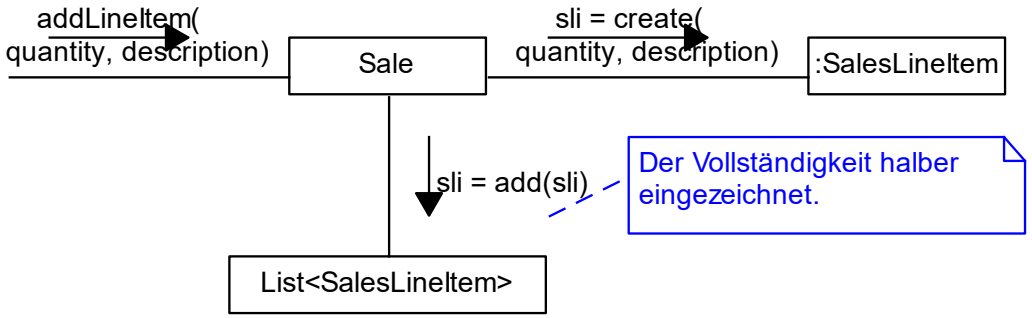


Warum braucht es ProductCatalog?

- Es ginge natürlich auch ohne diese Klasse.
- Domänen-Orientierung und High Cohesion sprechen für diese Klasse.
- Controller nicht überladen!

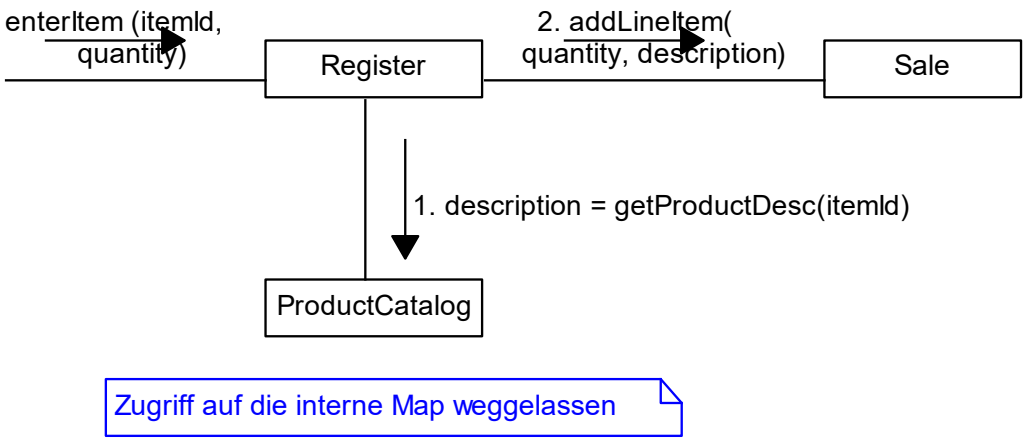
EnterItem (3)

1. Controller: Bereits definiert, nämlich Register
2. Ziel: Neue Instanz von SalesLineItem
   1. Creator Pattern anwenden: Container von SalesLineItem soll diese erstellen, das wäre die Klasse Sale.
   2. Bedingt neue Methode in Sale, die neben dem Erstellen einer neuen Instanz von SalesLineItem, diese der internen Liste hinzufügt.
   3. Bei der Instanzierung von SalesLineItem können die Attributewerte «quantity» und «description» gleich als Parameter mitgegeben werden.



EnterItem (4)

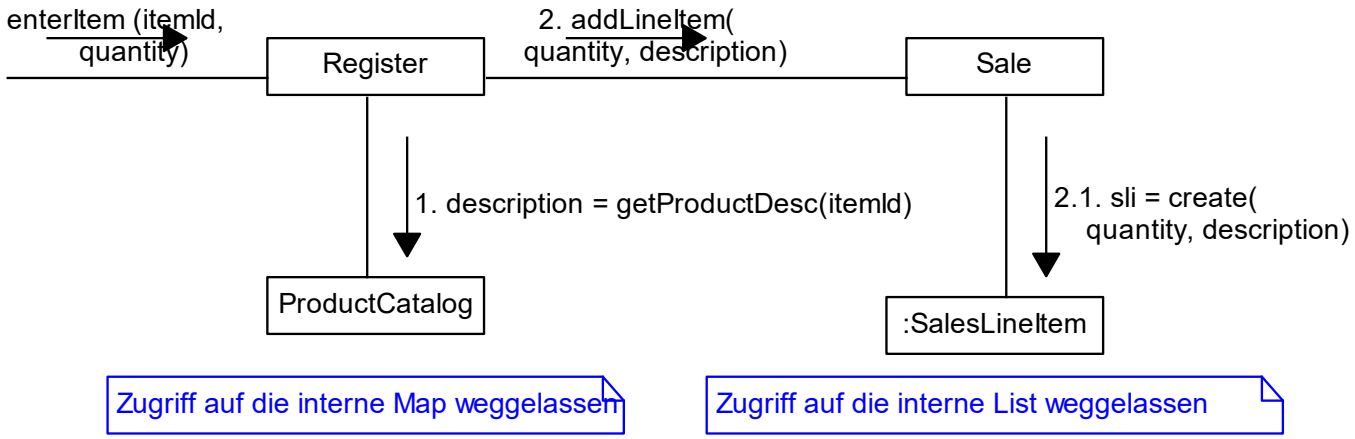
3. Weg zum Ziel:
   1. Sale ist direkt von Register aus erreichbar, d.h. wir brauchen keine Zwischenstation.
   2. Allerdings wird in SaleLineItem eine Referenz auf ProductDescription benötigt, die über den Parameter itemId spezifiziert wird.
   3. Bevor wir Sale den Auftrag geben, eine neue Instanz von SaleLineItem zu erstellen, müssen wir über den ProductCatalog die itemId in die entsprechende ProductSpecification Instanz umwandeln.



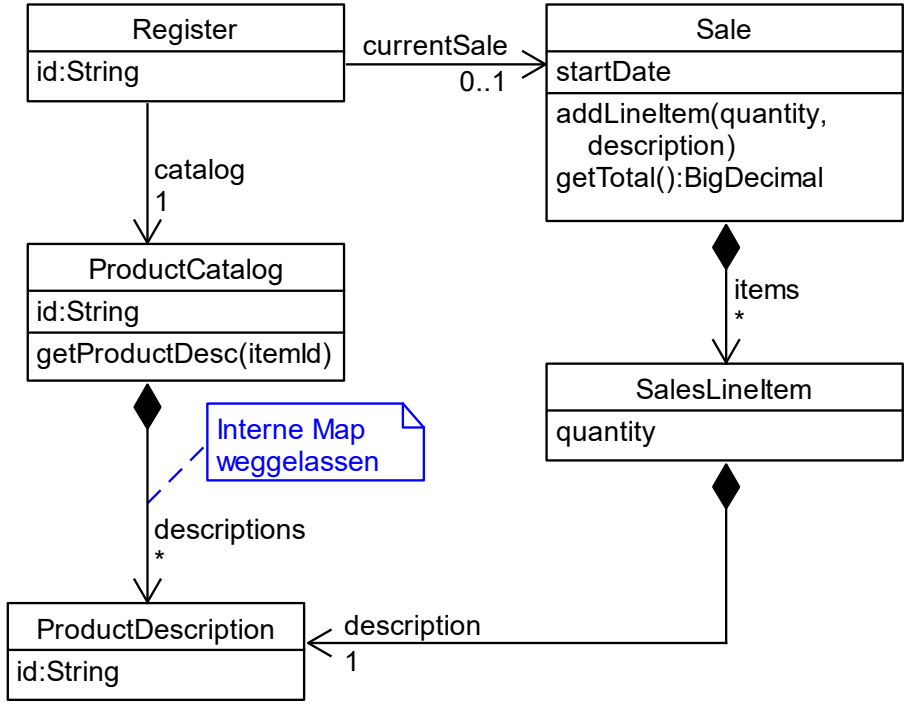
EnterItem (5)

4. Veränderungen programmieren: Bisherige Teile zusammenbauen

Use-Case Realization, dokumentiert mit Kommunikationsdiagramm



Resultierendes DCD



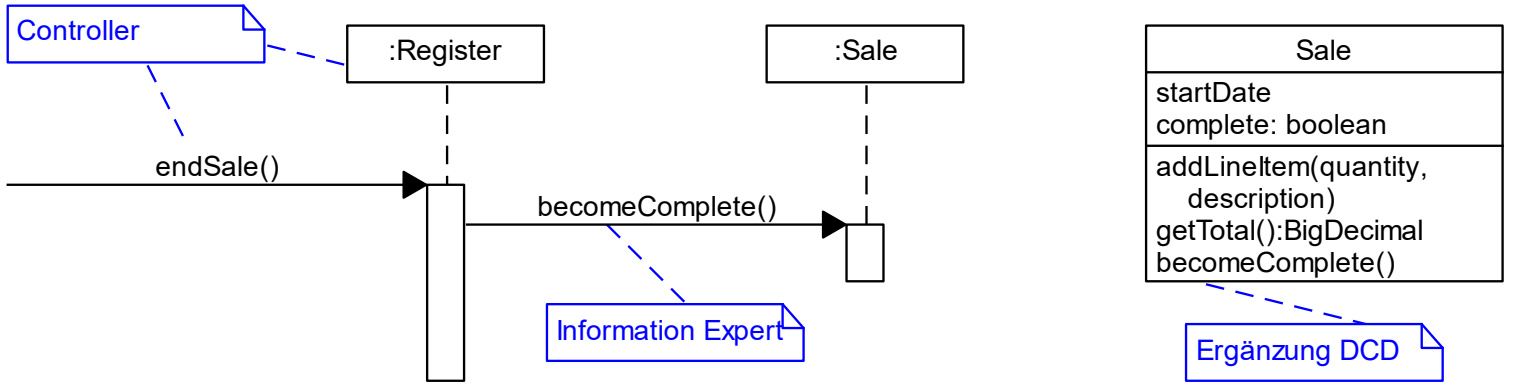
EndSale (1)

Vorbereitungsarbeiten

1. Use Case «Process Sale», fully dressed ausgearbeitet
2. Systemoperation «endSale()
3. Operation Contract, Nachbedingungen
   - `Aktuelle Sale`-Instanz ist markiert als „abgeschlossen“ (complete)
4. Aktueller Status der Software: Siehe vorhergehendes Beispiel
   1. Klassen Register, Sale erstellt
5. Die bereits existierenden Software-Klassen genügen für diese Systemoperation.

EndSale (2)

1. Controller: Bereits definiert, nämlich Register
2. Ziel: Sale als abgeschlossen markieren
3. Sale ist direkt von Register aus erreichbar
4. Sale erhält eine neue Methode: becomeComplete(). Im Moment setzt diese Methode nur ein boolean Attribute.



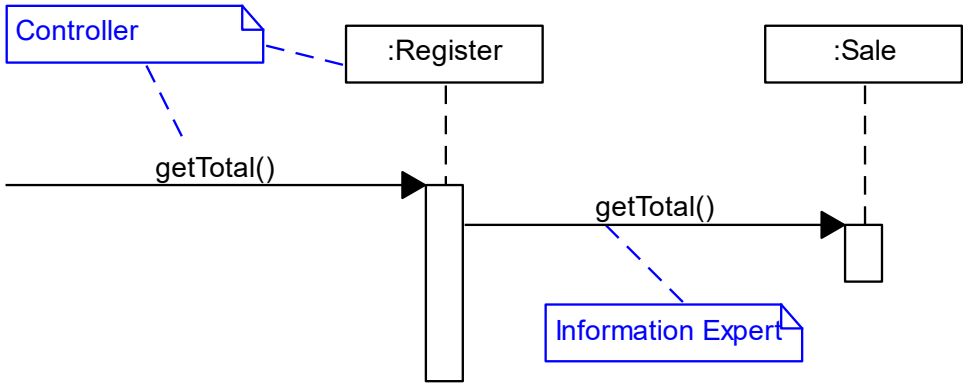
GetTotal (1)

Vorbereitungsarbeiten

1. Use Case «Process Sale», fully dressed ausgearbeitet
2. Systemoperation «getTotal()»
3. Operation Contract, Nachbedingungen
   - Reine Abfrage, macht keine Veränderungen
4. Aktueller Status der Software: Siehe vorhergehendes Beispiel
   1. Klassen Register, Sale, SalesLineItem und ProductDescription vorhanden
5. Die bereits existierenden Software-Klassen genügen für diese Systemoperation.

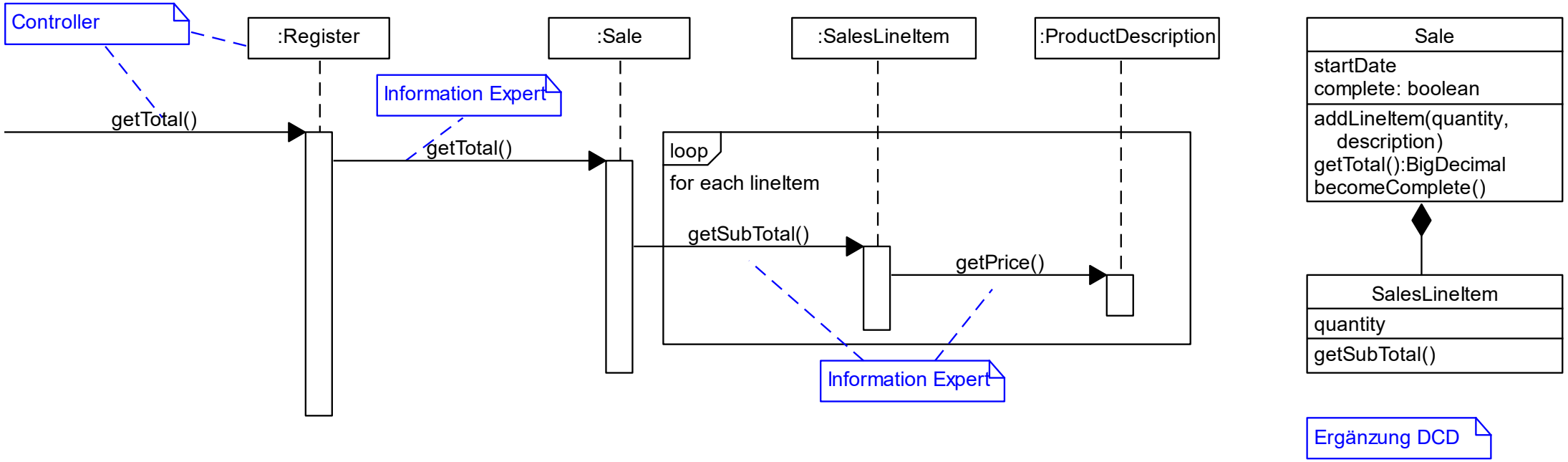
GetTotal (2)

1. Controller: Bereits definiert, nämlich Register
2. Ziel: Keine Veränderung, aber Rückgabewert zeigt den Gesamtbetrag der aktuellen Sale Instanz.
3. Sale ist der Information Expert (GRASP) für diese Aufgabe und direkt von Register aus erreichbar
4. Sale erhält eine neue Methode: getTotal(). Diese Verantwortlichkeit kann Sale aber nicht alleine wahrnehmen, es braucht noch SalesLineItem und ProductDescription dafür.



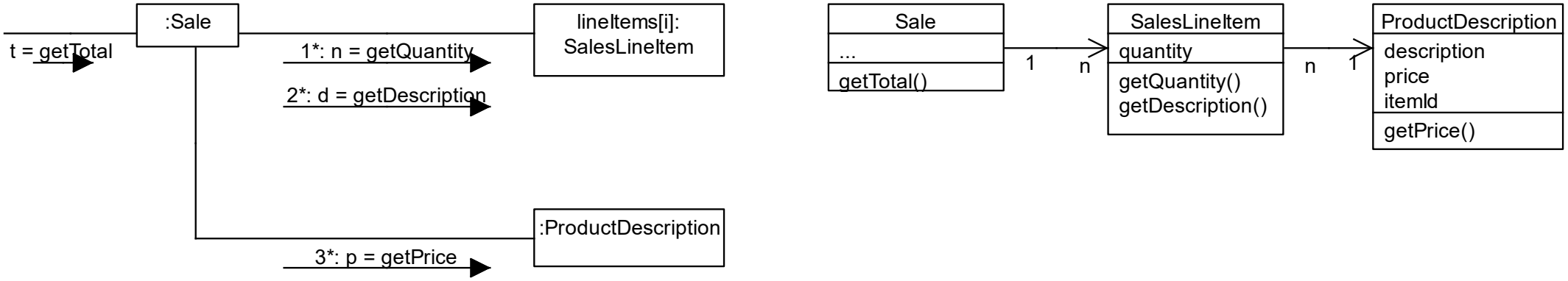
GetTotal (3)

5. Sale muss nun das Zwischentotal seiner SalesLineItems zusammenzählen und zurückgeben.
   – Die Berechnung des Zwischentotals wird gemäss Information Expert an SalesLineItem delegiert. Dafür ruft diese die Methode getPrice() von ProductDescription auf.



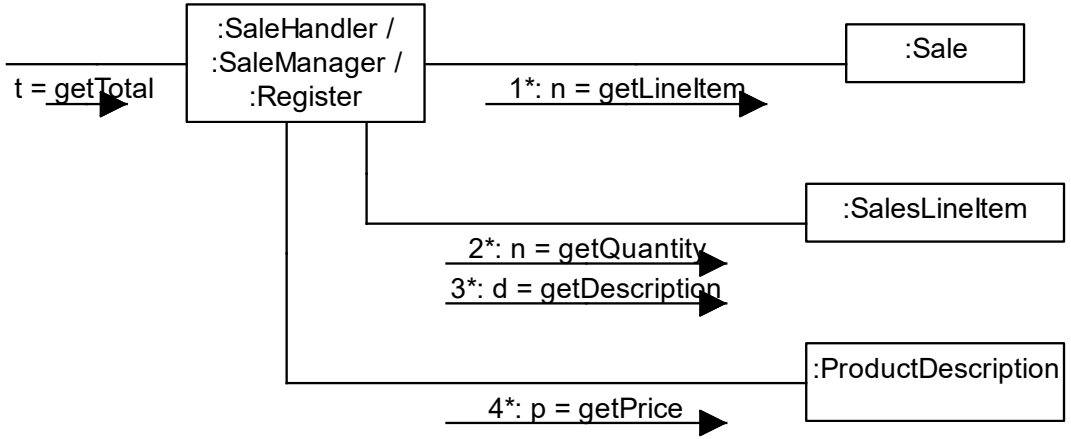
GetTotal (4)

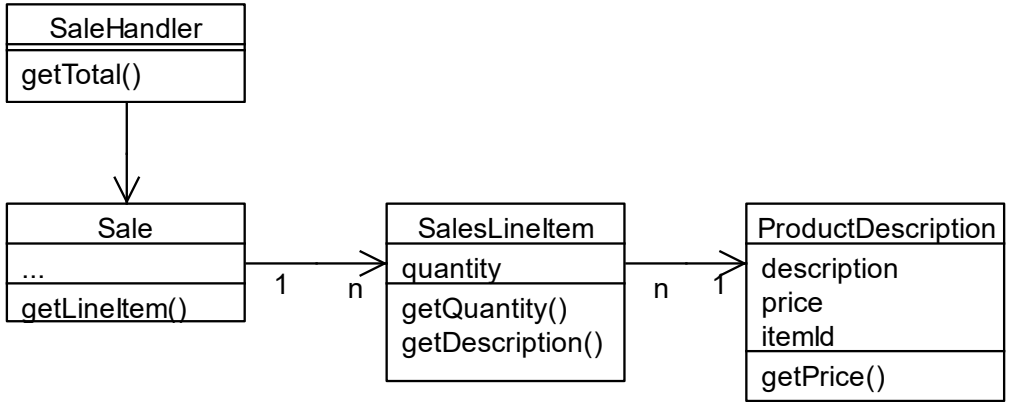
- Wir betrachten nun alternative Lösungen und evaluieren sie gemäss Low Coupling und High Cohesion.
- In der folgenden Variante macht Sale alles selber. Es holt sich alle Daten aus SalesLineItem und ProductDescription und führt alle Berechnungen selber durch.
- Dadurch sinkt die Kohäsion von Sale und das Information Expert Pattern ist für SalesLineItem nicht mehr erfüllt. Zusätzlich wird eine weitere Kopplung von Sale zu ProductDescription eingeführt.
- Fazit: Nicht empfehlenswert.



GetTotal (5)

- In der folgenden Variante führen wir noch eine weitere Klasse SaleHandler ein, die alle Berechnungen ausführt.
- Das Information Expert Pattern ist für Sale und SalesLineItem nicht mehr erfüllt.
- Zusätzliche Klasse und zusätzliche Kopplungen.
- Fazit: Nicht empfehlenswert.

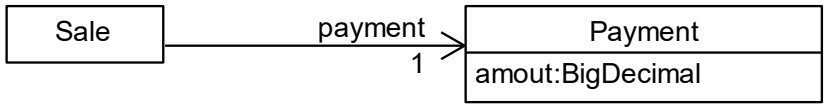




MakePayment (1)

Vorbereitungsarbeiten

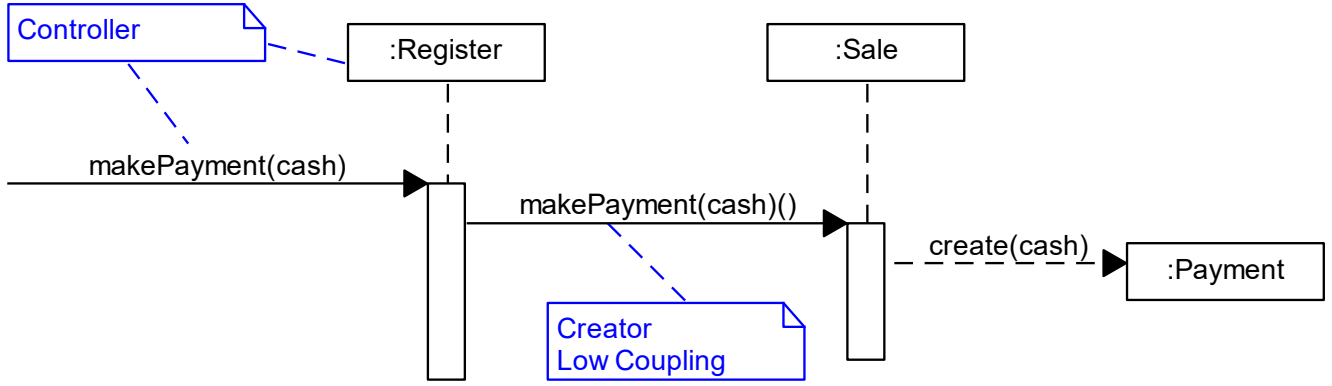
1. Use Case «Process Sale», fully dressed ausgearbeitet
2. Systemoperation «makePayment()»
3. Operation Contract, Nachbedingungen
   - Eine neue Instanz von Payment ist erzeugt und mit der aktuellen Instanz von Sale verknüpft.

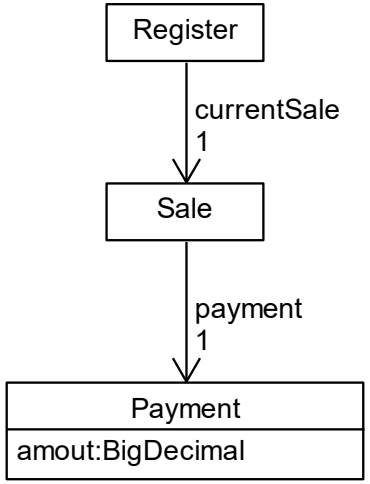


4. Aktueller Status der Software: Siehe vorhergehendes Beispiel
   1. Klassen Register, Sale, SalesLineItem und ProductDescription vorhanden
5. Wir brauchen eine Software-Klasse Payment, die wir aus dem Domänenmodell übernehmen können.
   - Vielleicht erscheint es übertrieben, dafür eine eigene Software-Klasse mit nur einem Attribut zu erstellen.
     - Die Orientierung an der Fachdomäne empfiehlt aber das Einführen auch „kleiner“ Software Klassen, die aus der Fachdomäne inspiriert sind
   - Die Bezahlung mit Cash ist eine Variante. In späteren Iterationen kommt die Bezahlung mit Kreditkarte hinzu.

MakePayment (2)

1. Controller: Bereits definiert, nämlich Register
2. Ziel: Neue Instanz von Payment, mit der aktuellen Instanz von Sale verknüpft.
3. Payment ist über Sale von Register aus erreichbar.
4. Wer erzeugt die neue Instanz von Payment?
   1. Creator Pattern anwenden: Sale ist nur bedingt der Container von Payment, arbeitet aber sehr eng damit zusammen.
   2. Wichtiger ist allerdings, dass alternative Varianten wegen Low Coupling schlechter abschneiden.

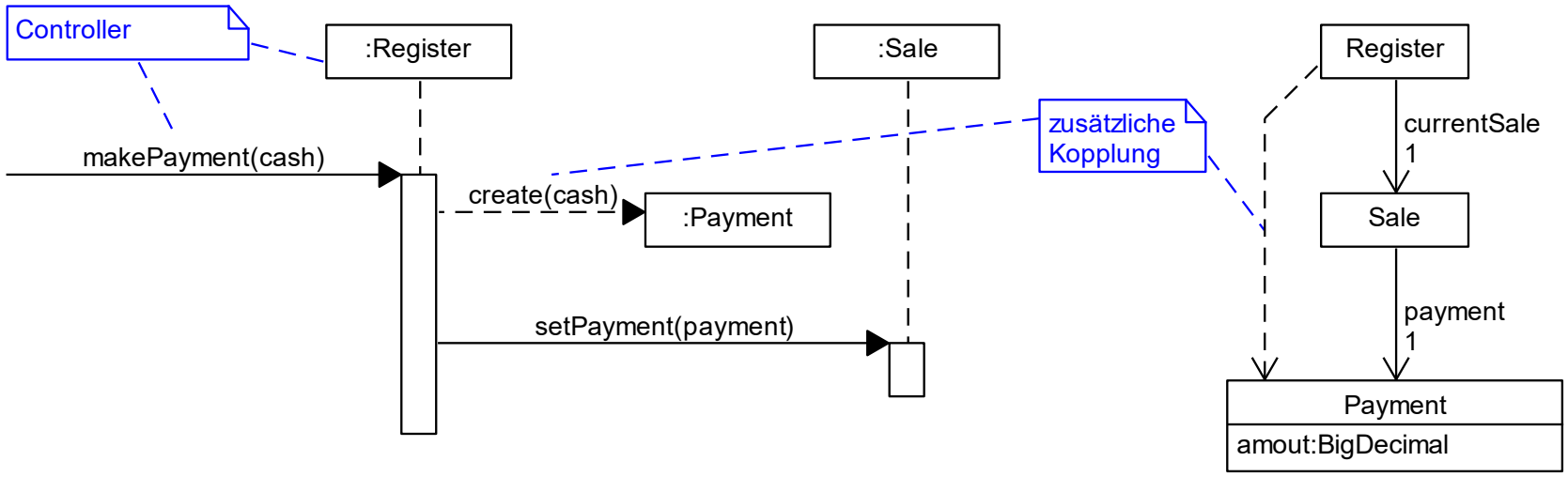




MakePayment (3)

Alternative Lösung:

- Payment wird von Register erzeugt und an Sale übergeben.
- Dadurch wird eine neue Kopplung von Register auf Payment eingeführt, weshalb diese Lösung nicht empfehlenswert ist.



Agenda

1. Einfluss Analyse Artefakte
2. UML und Design to Code
3. Repetition GRASP
4. Vorgehen
5. Fallstudie «NextGenPos»
6. Fallstudie «Monopoly»
7. Wrap-up und Ausblick

Einleitung

- Wir führen nun die Use-Case Realization für die erste Iteration zur Entwicklung des Spiels „Monopoly“ durch. Es ist ein weiteres Fallbeispiel aus Larman[1]
- In der ersten Iteration spielt der Computer „mit sich selber“. Der Mensch beschränkt sich auf das Initialisieren und den Start des Spiels
- Die Systemoperation:
  - initialize(numOfPlayers)
  - playGame(idemId, quantity)
- Pro (künstlichem) Spieler muss zuerst mit 2 Würfeln gewürfelt werden, dann wird seine Spielfigur um die Anzahl Augen verschoben.
- In der ersten Iteration passiert noch nichts, wenn eine Figur auf einem Spielfeld landet.

Agenda

1. Einfluss Analyse Artefakte
2. UML und Design to Code
3. Repetition GRASP
4. Vorgehen
5. Fallstudie «NextGenPos»
6. Fallstudie «Monopoly»
7. Wrap-up und Ausblick

Wrap-up

- Use-Case-Realization heisst die Tätigkeit, einen Use-Case zu realisieren, indem Software-Klassen Verantwortlichkeiten bis hin zum Code erhalten.
- Use-Case-Realization basiert auf den Analyse-Artefakten aus LE03 und LE04.
- Die Software Architektur gibt Vorgaben für die Umsetzung der Use-Cases.
- Die Anwendung der GRASP Prinzipien führt zu gutem, objektorientiertem und wartbarem Code.

Ausblick

- In der nächsten zwei Lerneinheiten werden wir:
  - die GoF Design Patterns einführen und an vielen praktischen Beispielen anwenden.

Quellenverzeichnis

[1] Larman, C.: UML 2 und Patterns angewendet, mitp Professional, 2005