Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InIT Institut für angewandte
Informationstechnologie

Bachelor of Science (BSc) in Informatik

Modul Software-Entwicklung 1 (SWEN1)

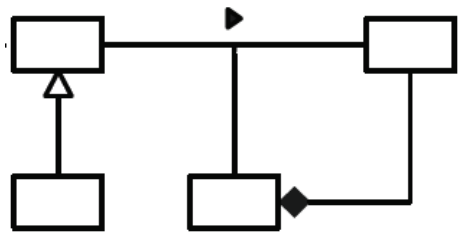
LE 06 – Softwarearchitektur und Design II

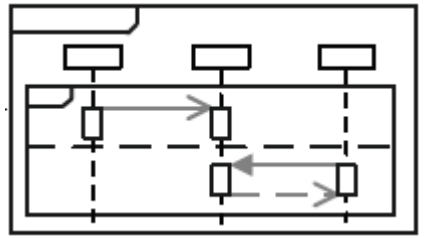
SWEN1/PM3 Team:
R. Ferri (feit), D. Liebhart (lieh), K. Bleisch (bles), G. Wyder (wydg)

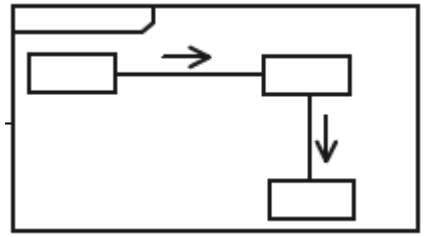
Ausgabe: HS24

Um was geht es?

- Wie realisiere ich einen Use Case mit Klassen, die klare Verantwortlichkeiten haben, wartbar und einfach erweiterbar sind?
- Wie modelliere ich mein Design mit der UML, um es diskutieren und evaluieren zu können?







Lernziele LE 06 – Softwarearchitektur und Design II

- Sie sind in der Lage:
  - den Zweck und die Anwendung von statischen und dynamischen Modellen im Design zu erläutern,
  - einen Objektentwurf zweckmässig mit UML-Klassen-, UML-Interaktions-, UML-Zustands- und UML-Aktivitätsdiagrammen darzustellen.
  - die Idee von Verantwortlichkeiten und des Responsibility-Driven Designs (RDD) für den Entwurf von Klassen zu erklären,
  - grundlegende Prinzipen und Pattern für den Klassenentwurf anzuwenden (GRASP, SOLID),

Agenda

1. Einführung in das objektorientierte Design
2. UML-Diagramme für das Design
3. Klassen mit Verantwortlichkeiten entwerfen
4. Wrap-up und Ausblick

Recap: Objektorientierung

- Was bedeutet Objektorientierung?

Recap: Objektorientierte Analyse und objektorientiertes Design

domain concept

Plane
tailNumber

visualization of domain concept

representation in an object-oriented programming language

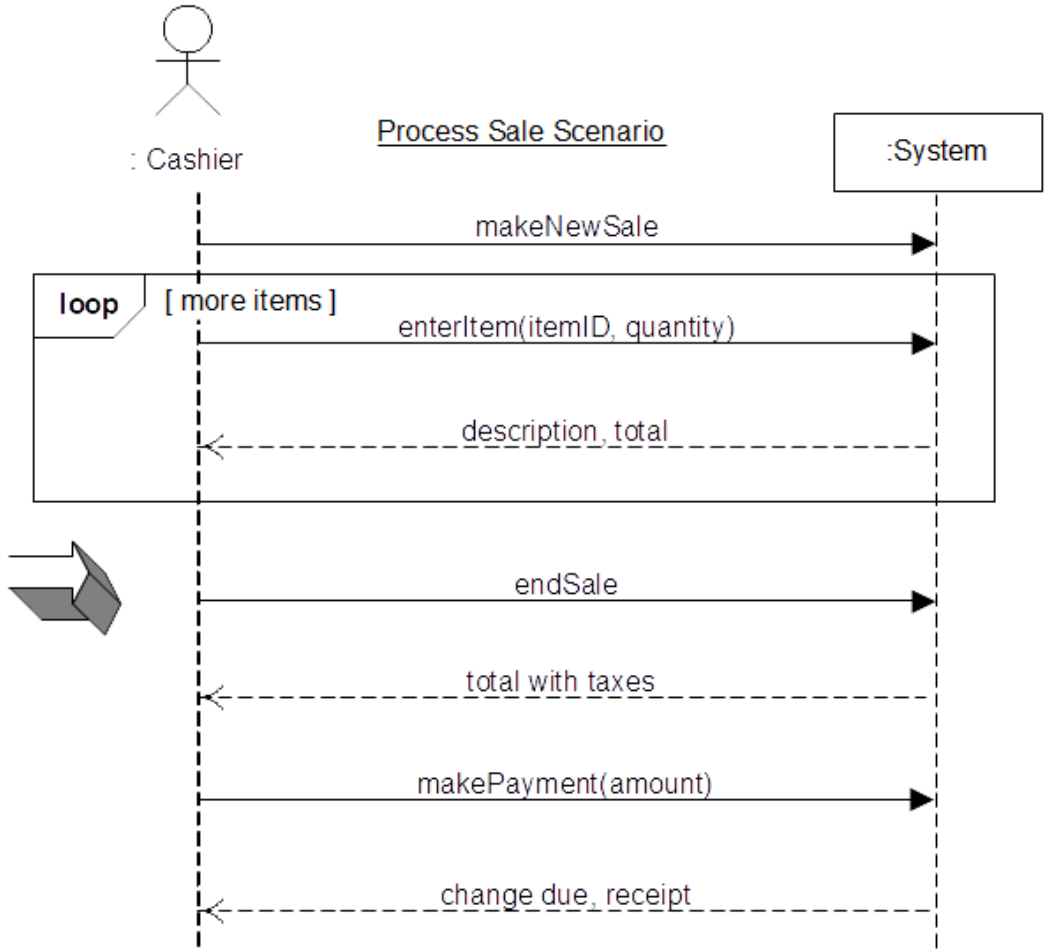
```
public class Plane
{
private String tailNumber;

public List getFlightHistory() {...}
}
```

Use Cases und System-Sequenzdiagramm (SSD)

- Szenarien und Systemoperationen, die in den Use Cases identifiziert wurden, zusammen mit dem Domänenmodell bilden die Basis für das Design.
- Die Systemoperation bzw. deren Antworten sind schlussendlich das, was programmiert werden muss.

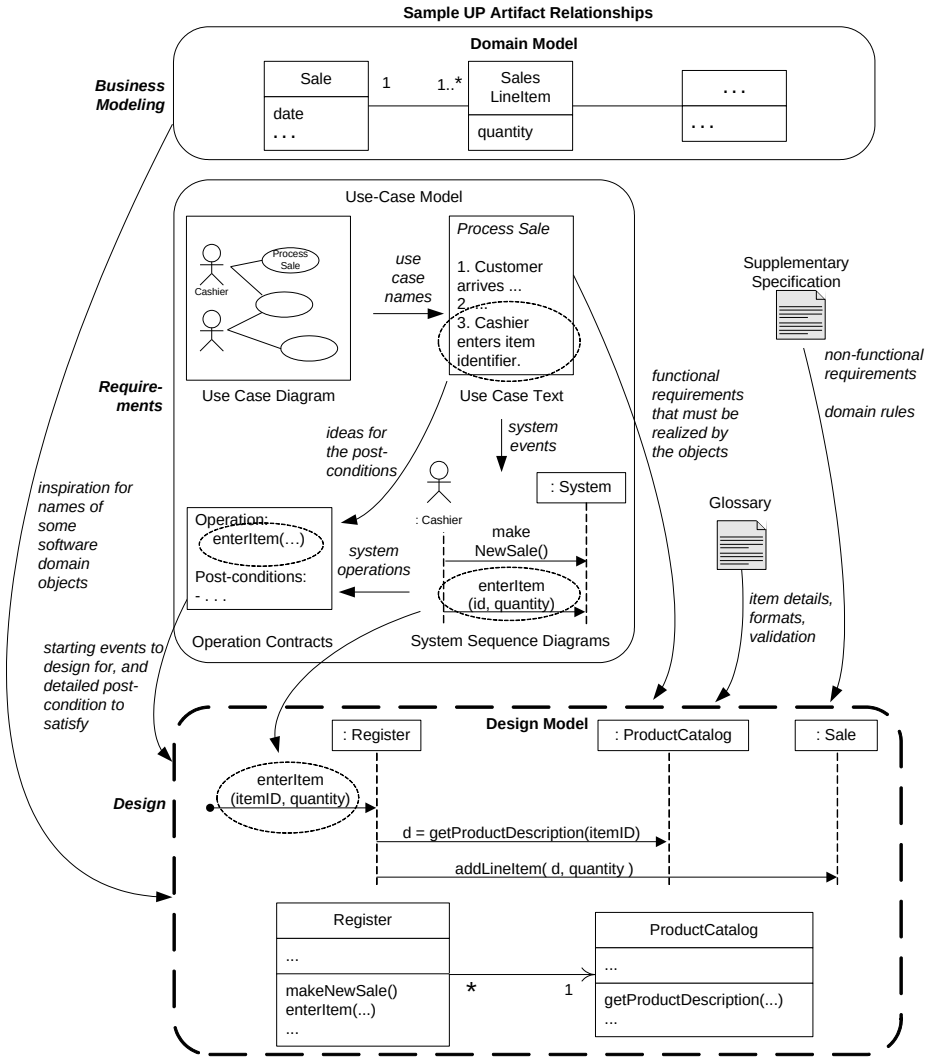
Simple cash-only Process Sale scenario:

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.
...



Use Cases und Use-Case-Realisierung

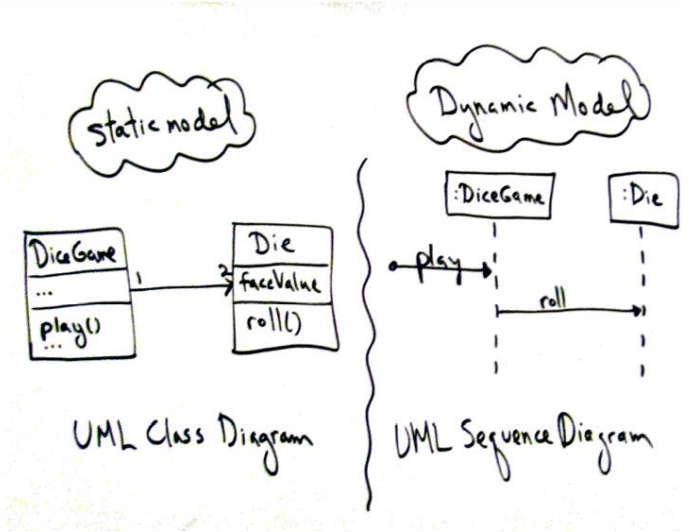
- Eine Use-Case-Realisierung beschreibt, wie ein bestimmter Use Case innerhalb des Designs mit kollaborierenden Objekten realisiert wird.
- Jedes Szenario eines Use Cases bzw. dessen Systemoperationen werden schrittweise entworfen und implementiert.
- Die UML-Diagramme sind eine gemeinsame Sprache, um Use-Case-Realisierungen zu veranschaulichen und zu diskutieren.



Klassen entwerfen: Statische und dynamische Modellierung

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

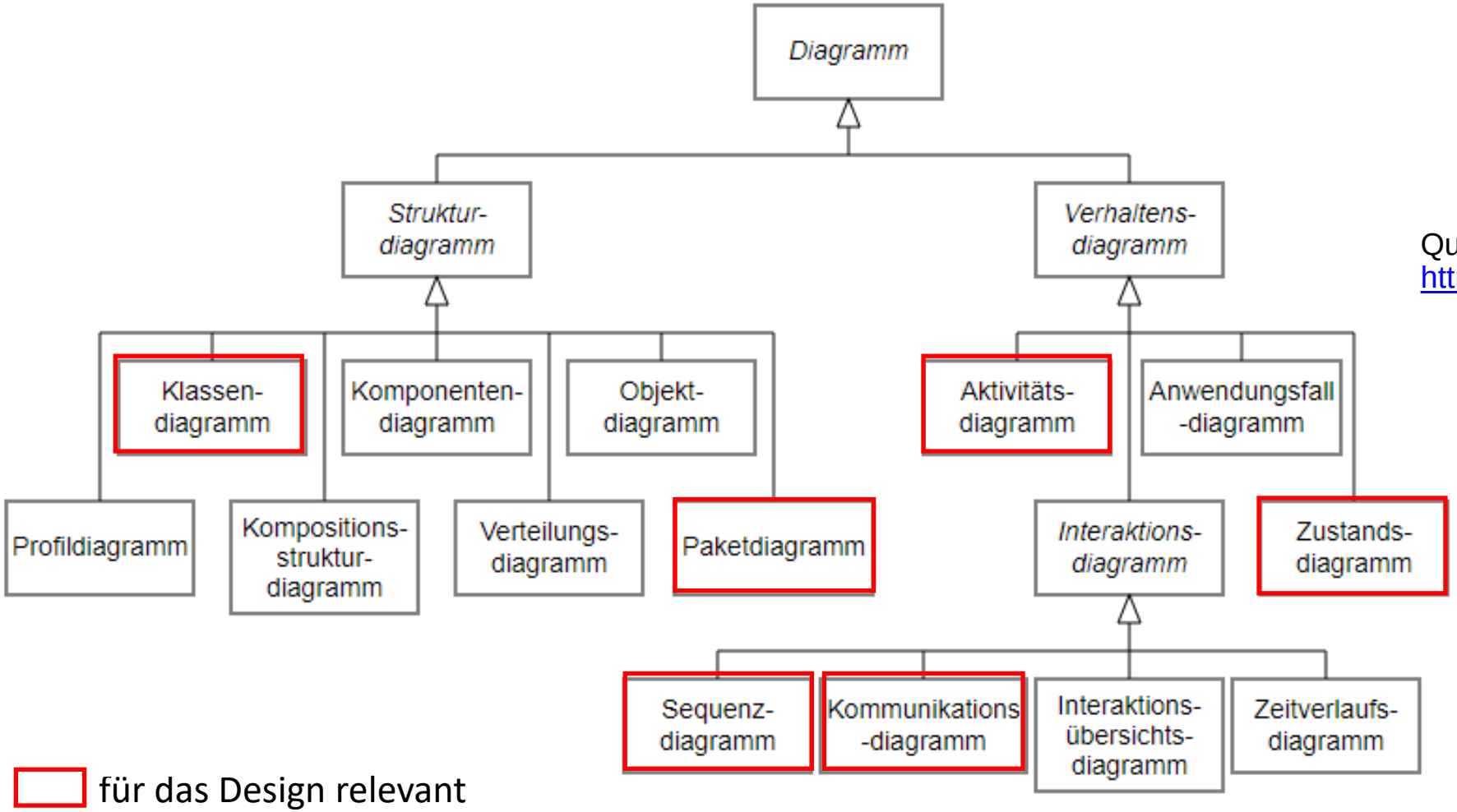
- Es gibt zwei Arten von Design-Modellen:
  - **Statische Modelle**
    - Statische Modelle, wie beispielsweise das UML-Klassendiagramm, unterstützen den Entwurf von Paketen, Klassennamen, Attributen und Methodensignaturen (ohne Methodenkörper).
  - **Dynamische Modelle**
    - Dynamische Modelle, wie beispielsweise UML-Interaktionsdiagramme, unterstützten den Entwurf der Logik, des Verhaltens des Codes und der Methodenkörper.
- Statische und dynamische Modelle ergänzen sich.
- Statische und dynamische Modelle werden parallel erstellt.



Agenda

1. Einführung in das objektorientierte Design
2. **UML-Diagramme für das Design**
3. Klassen mit Verantwortlichkeiten entwerfen
4. Wrap-up und Ausblick

Die Diagramme der UML

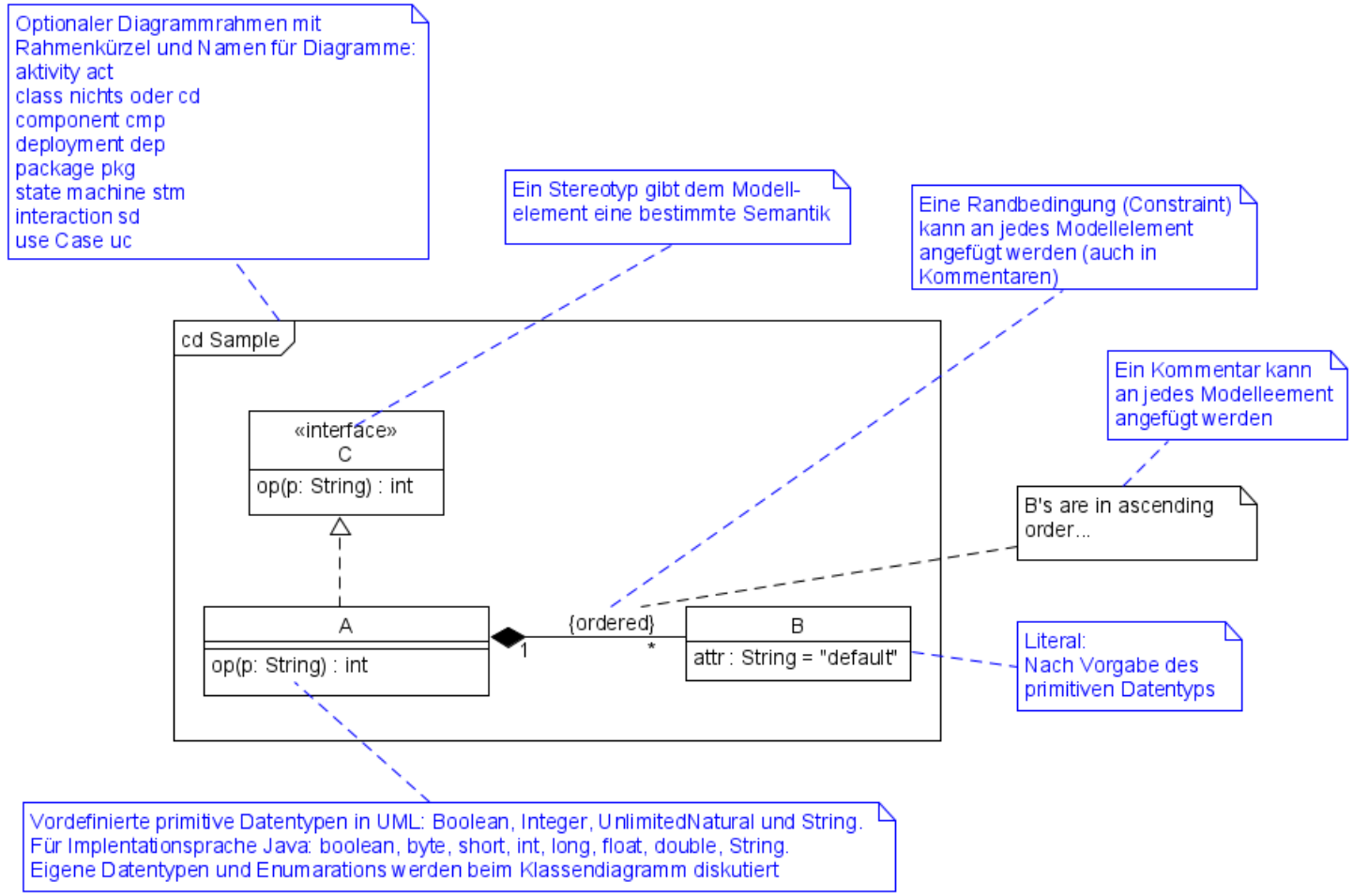


für das Design relevant

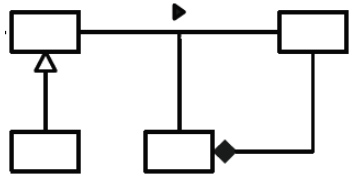
Quelle: UML Specification, https://www.omg.org/spec/UML/

Grundelemente der UML

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



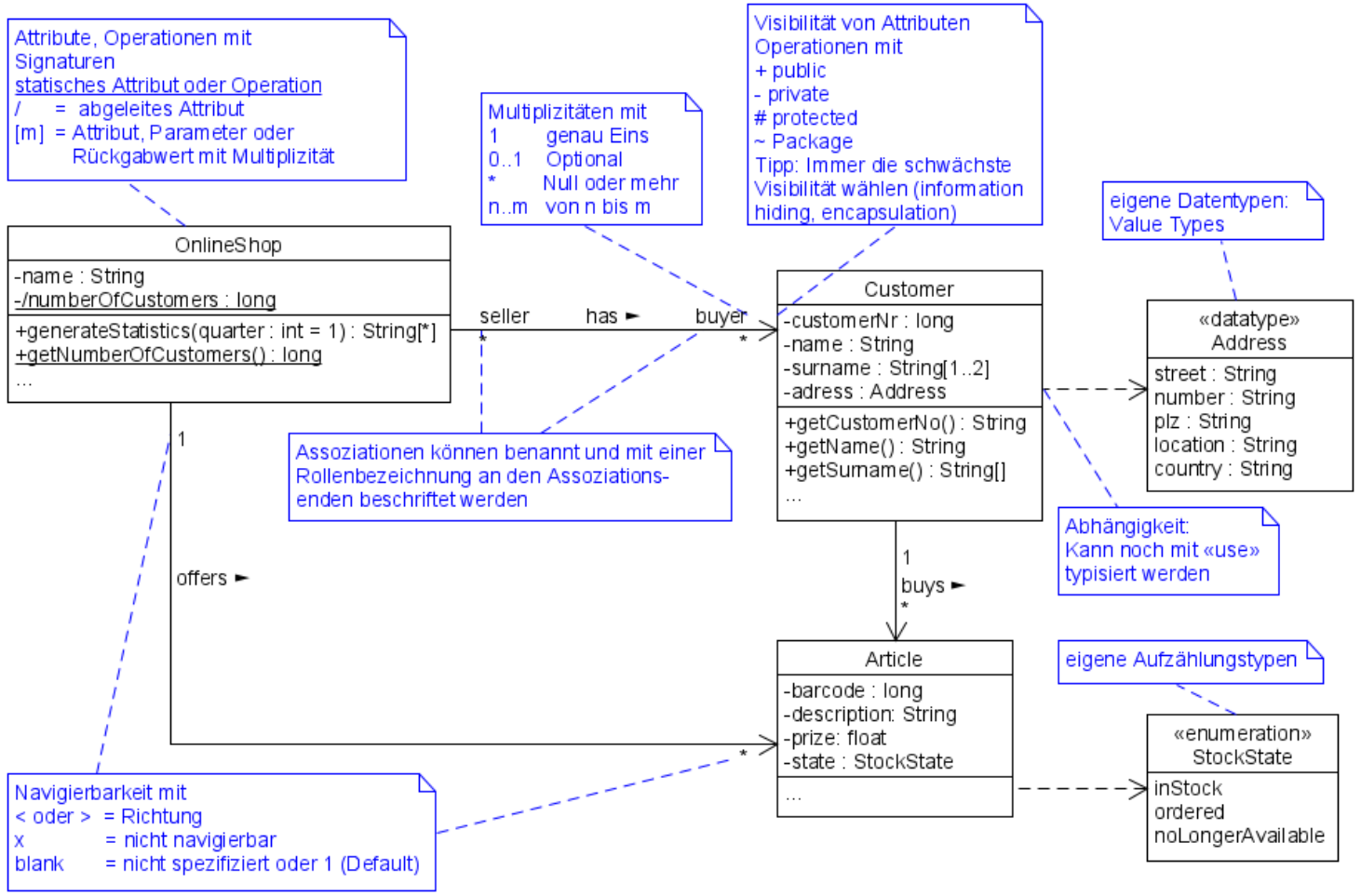
UML-Klassendiagramm (1/7)



- Das Diagramm beantwortet die zentrale Frage:
  Aus welchen Klassen besteht mein System und wie sind sie miteinander verknüpft?
- Es beschreibt die statische Struktur des zu entwerfenden oder abzubildenden Systems.
  - Welche Klassen und Objekte existieren im System
  - Welche Attribute, Operationen und Beziehungen haben sie untereinander
  - Es enthält alle relevanten Strukturzusammenhänge und Datentypen.
- Es bildet die Brücke zwischen den dynamischen Diagrammen.
- Das UML-Klassendiagramm kann für mehrere Zwecke verwendet werden:
  - In der Konzeptphase als Domänenmodell mit einem vereinfachten UML-Klassendiagramm (Problemdomäne).
  - In der Designphase als Design-Klassendiagramm (DCD) mit zusätzlichen Notationselementen (Lösungsdomäne).

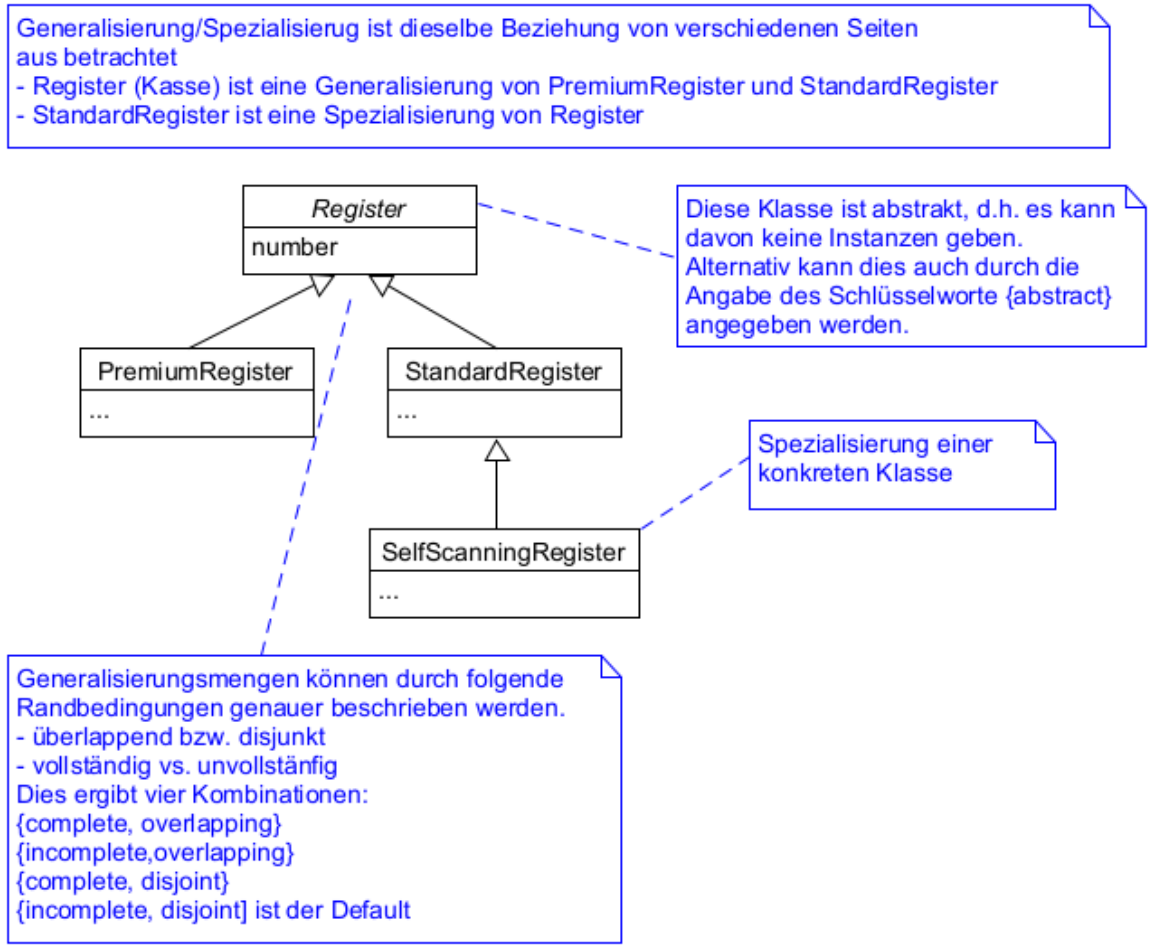
UML-Klassendiagramm (2/7)

- Notationselemente:
  - Klasse
  - Attribut
  - Operation
  - Sichtbarkeit von Attributen und Operationen
  - Assoziationsname, Rollen an den Assoziationsenden
  - Multiplizität
    - Bezieht sich auf die Objekte der betreffenden Klasse
  - Navigierbarkeit in Assoziationen
  - Datentypen und Enumerationen



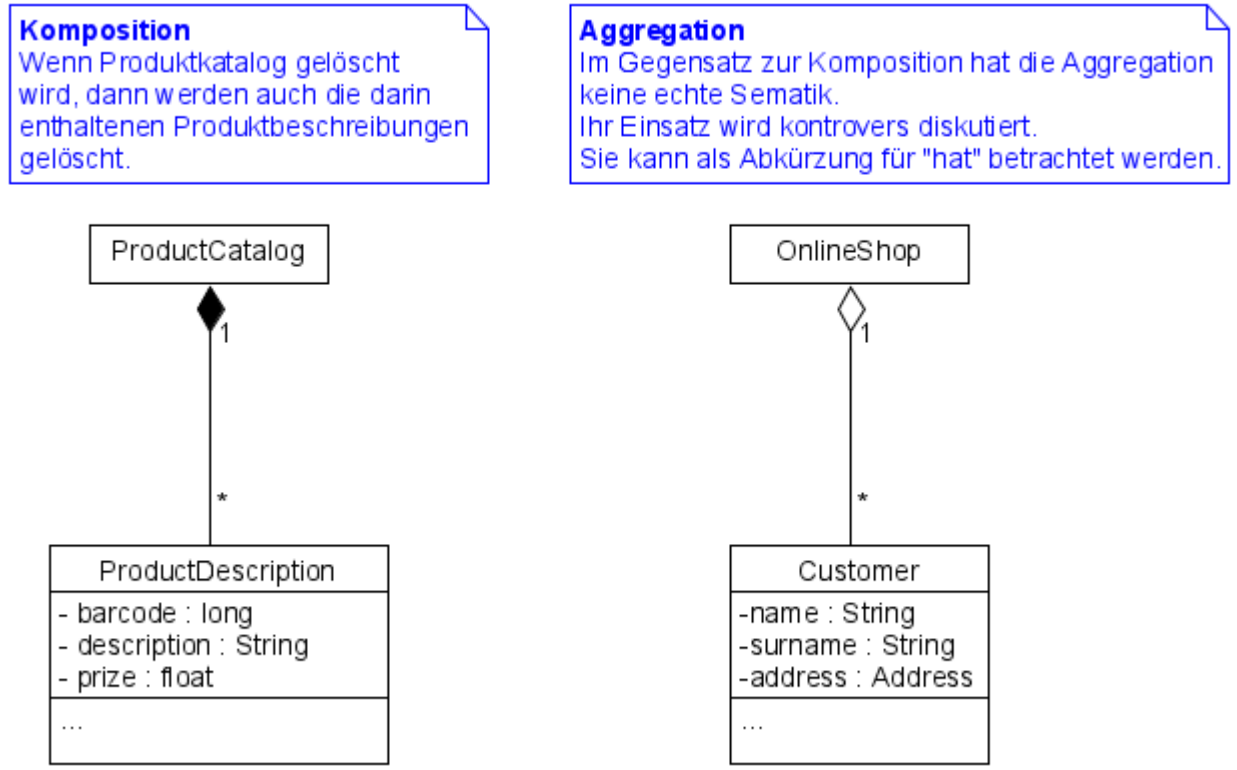
UML-Klassendiagramm (3/7)

- Notationselemente:
  - Generalisierung / Spezialisierung
  - Abstrakte Klassen



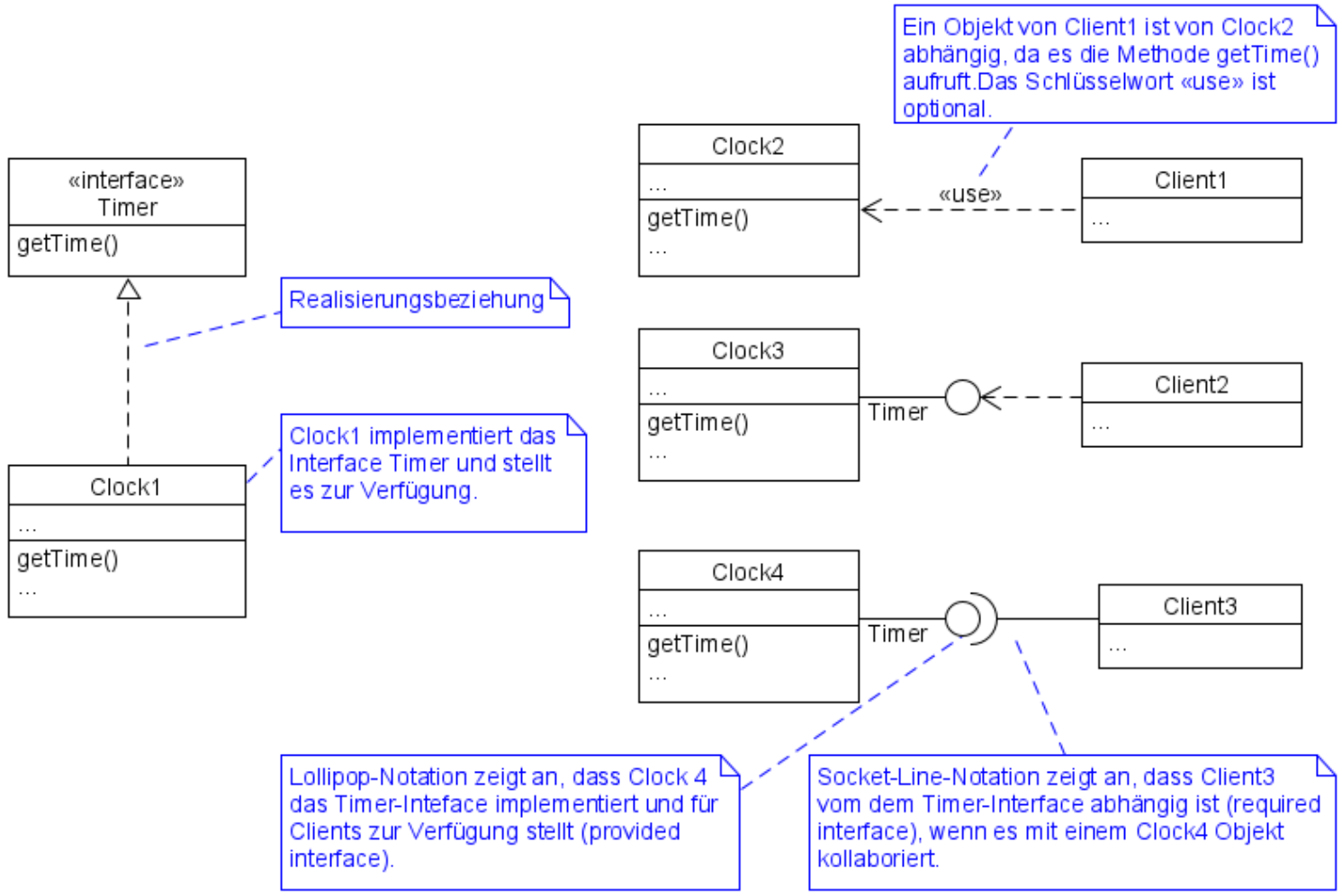
UML-Klassendiagramm (4/7)

- Notationselemente:
  - Komposition
  - Aggregation
- Komposition und Aggregation sind spezielle Assoziationen.



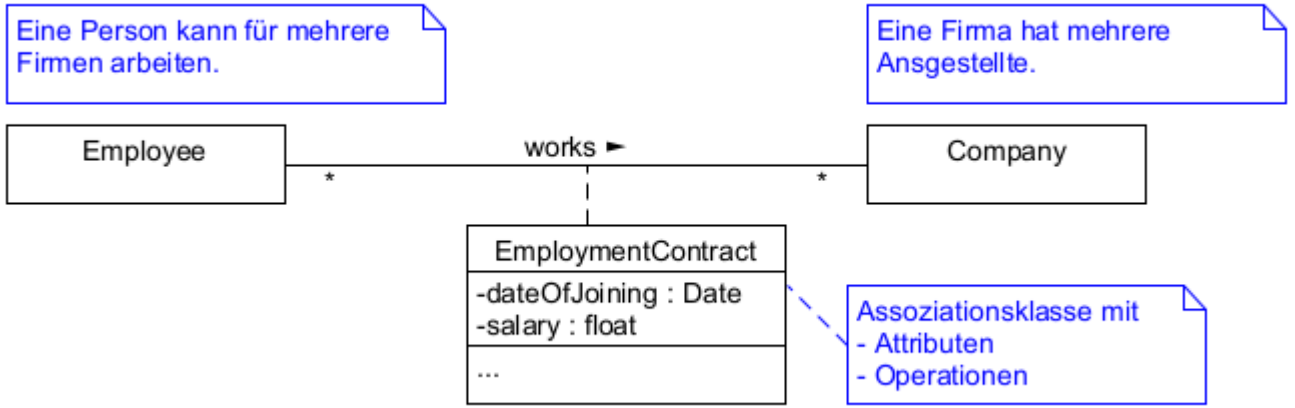
UML-Klassendiagramm (5/7)

- Notationselemente:
  - Interface
  - Interface-Realisierung
- Ein Interface beschreibt eine Menge von öffentlichen Operationen, Merkmalen und «Verpflichtungen», die durch eine Klasse, die die Schnittstelle implementiert, zwingend zur Verfügung gestellt werden müssen.



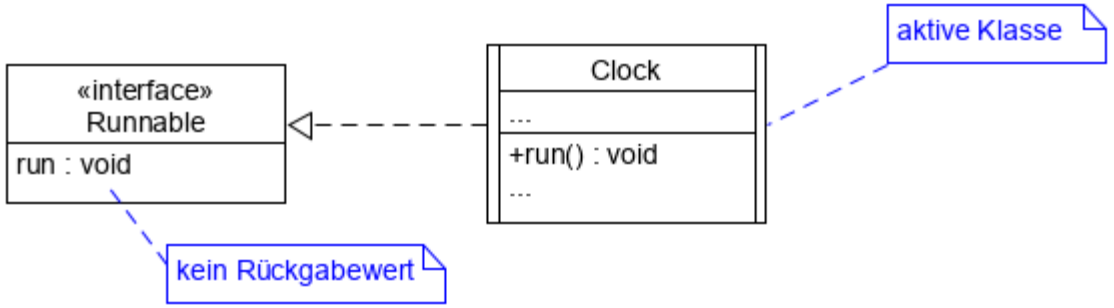
UML-Klassendiagramm (6/7)

- Notationselement: Assoziationsklasse
- Ein eigenes Modellelement, das die Eigenschaften der Klasse und der Assoziation in sich vereinigt.
- Ist mit einer existierenden Assoziation verbunden.
- Anwendung:
  - Attribute sind mit der Assoziation verbunden
  - Lebensdauer der Instanzen der Assoziationsklasse von der Assoziation abhängig
  - Viele-zu-viele-Assoziation zwischen den Klassen, die mit der Assoziation selbst verbunden sind



UML-Klassendiagramm (7/7)

- Notationselement: Aktive Klasse
- Eine Instanz einer aktiven Klasse wird in einem separaten Thread ausgeführt.

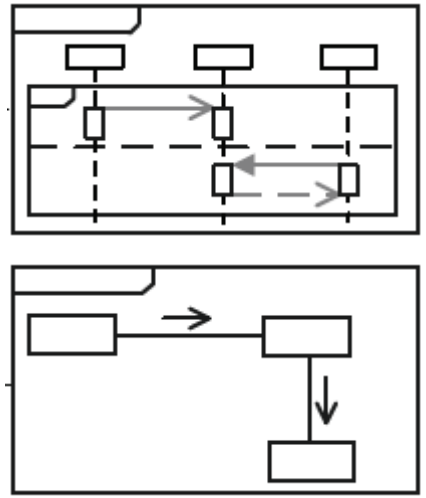


UML-Interaktionsdiagramme

- Ein Interaktionsdiagramm spezifiziert, auf welche Weise Nachrichten und Daten zwischen Interaktionspartnern ausgetauscht werden.

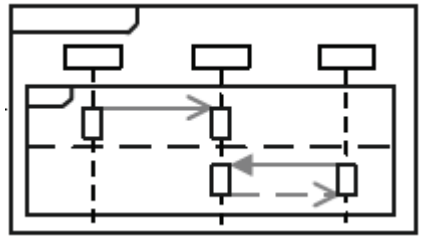
- Es gibt zwei Arten von UML-Interaktionsdiagrammen:
  - Sequenzdiagramm
  - Kommunikationsdiagramm

- Modellieren die Kollaborationen bzw. den Informationsaustausch zwischen Objekten (Dynamik).



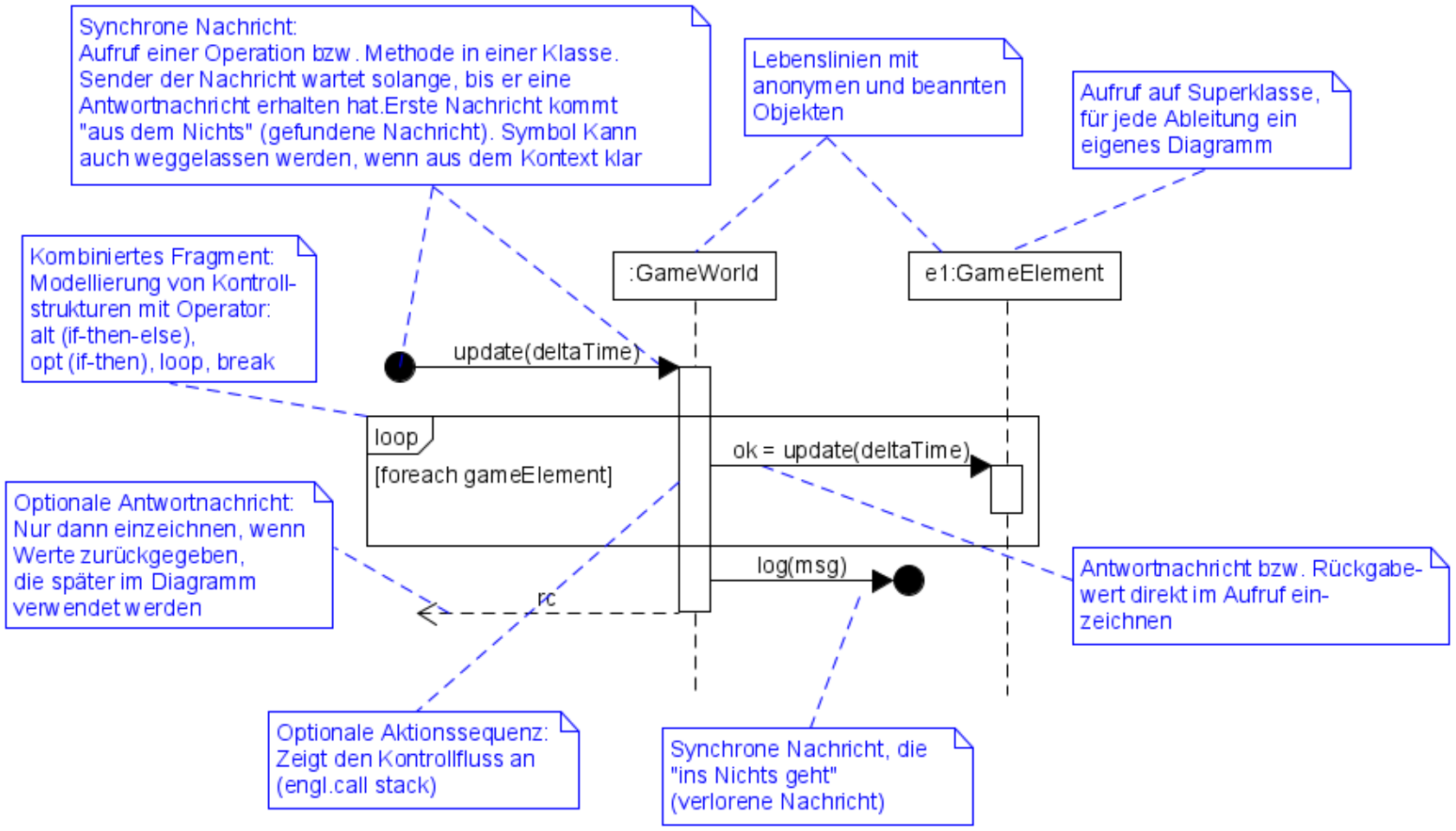
UML-Sequenzdiagramm (1/4)

- Das Diagramm beantwortet die zentrale Frage:
  Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?
- Es stellt den zeitlichen Ablauf des Informationsaustausches zwischen Kommunikationspartnern dar.
- Es sind Schachtelung und Flusssteuerung (Bedingungen, Schleifen, Verzweigungen) möglich.



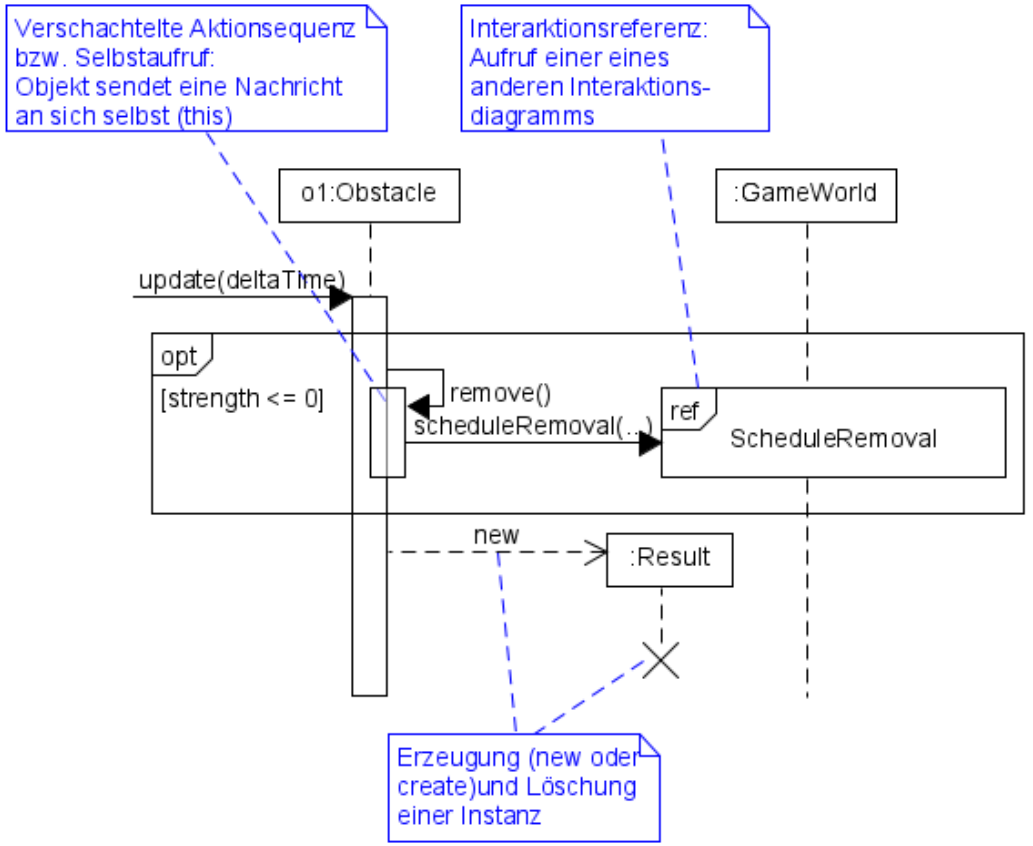
UML-Sequenzdiagramm (2/4)

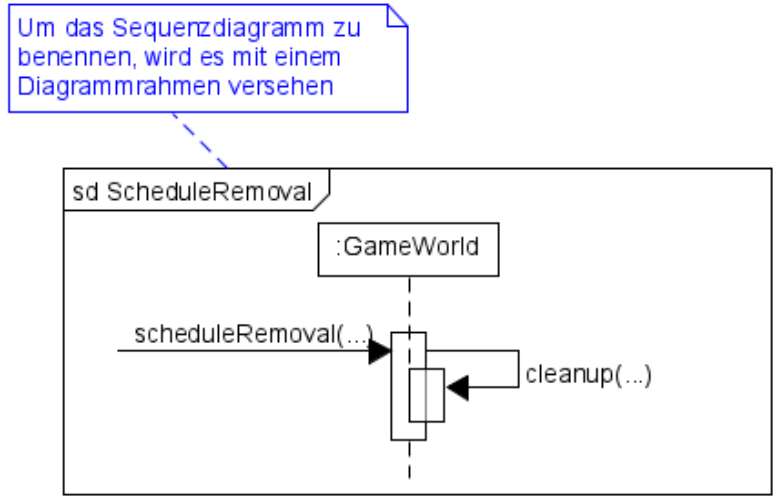
- Notationselemente:
  - Lebenslinie
  - Aktionssequenz
  - Synchrone Nachricht
  - Antwortnachricht
  - Gefundene, verlorene Nachricht
  - Kombiniertes Fragment



UML-Sequenzdiagramm (3/4)

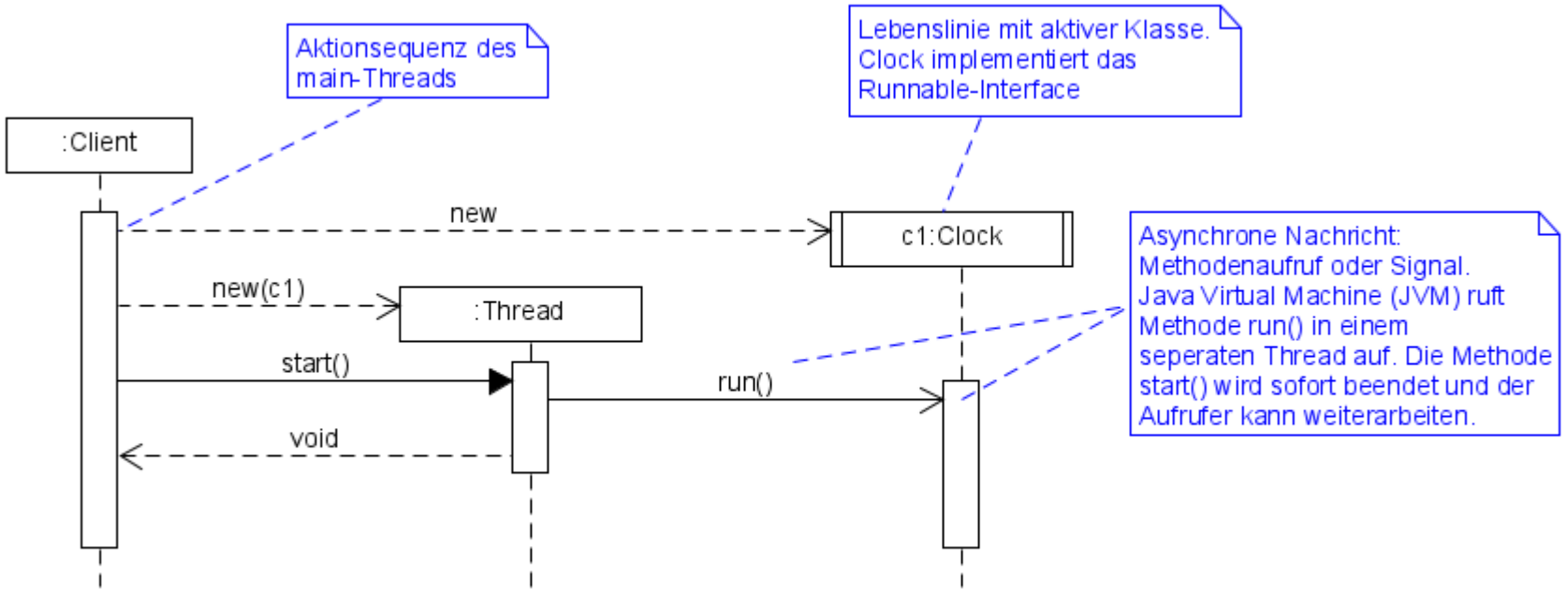
- Notationselemente:
  - Erzeugungs-, Löschereignis
  - Selbstaufruf
  - Interaktionsreferenz





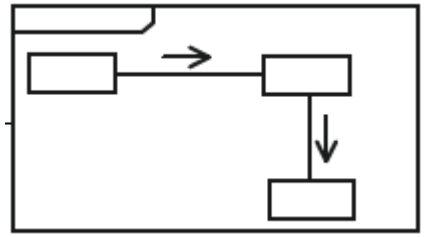
UML-Sequenzdiagramm (4/4)

- Notationselemente:
  - Lebensline mit aktiver Klasse
  - Asynchrone Nachricht



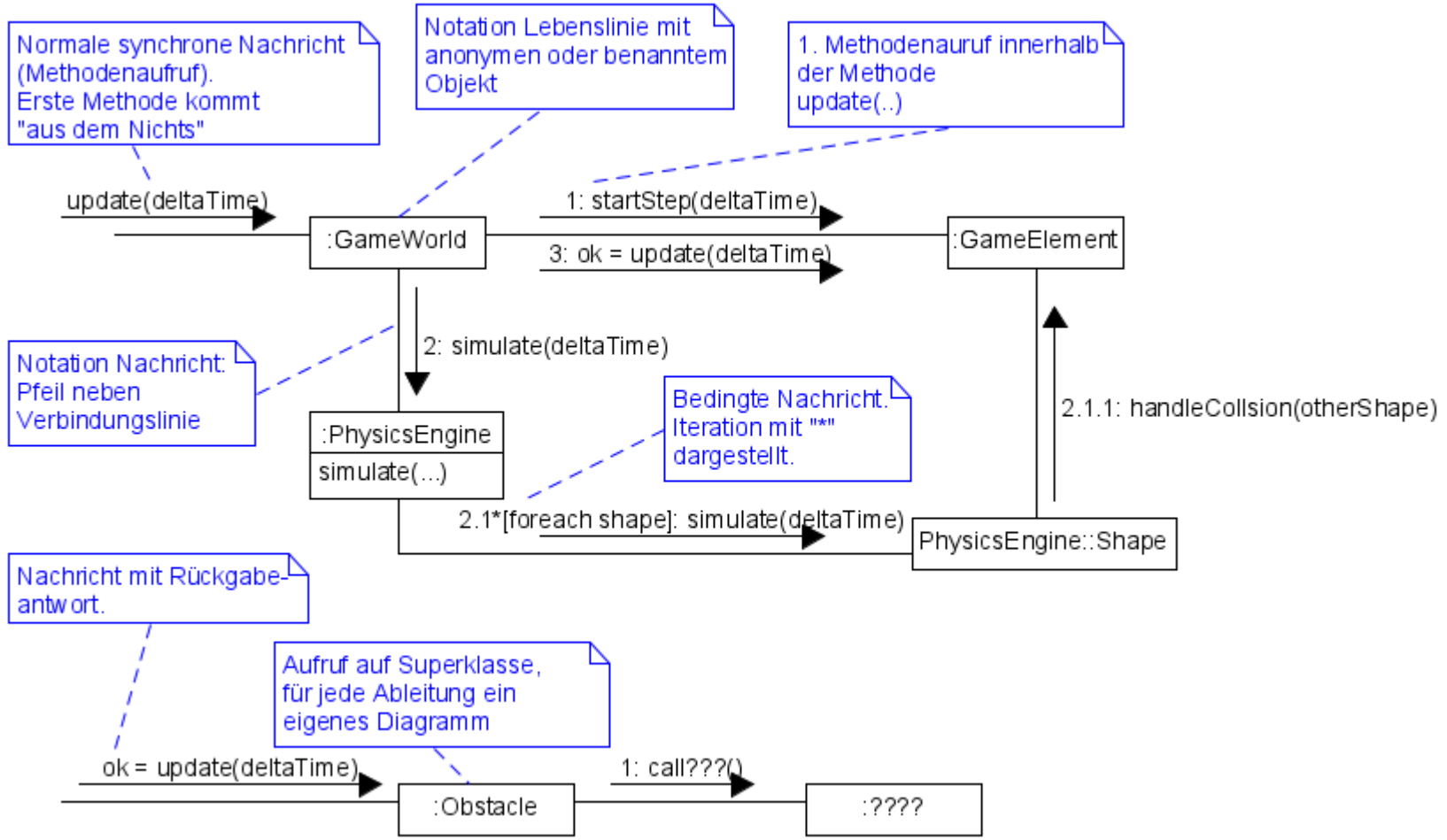
UML-Kommunikationsdiagramm (1/3)

- Das Diagramm beantwortet die zentrale Frage:
  Wer kommuniziert mit wem? Wer «arbeitet» im System zusammen?
- Es stellt ebenfalls den Informationsaustausch zwischen Kommunikationspartnern dar.
- Der Überblick steht im Vordergrund (Details und zeitliche Abfolge sind weniger wichtig).



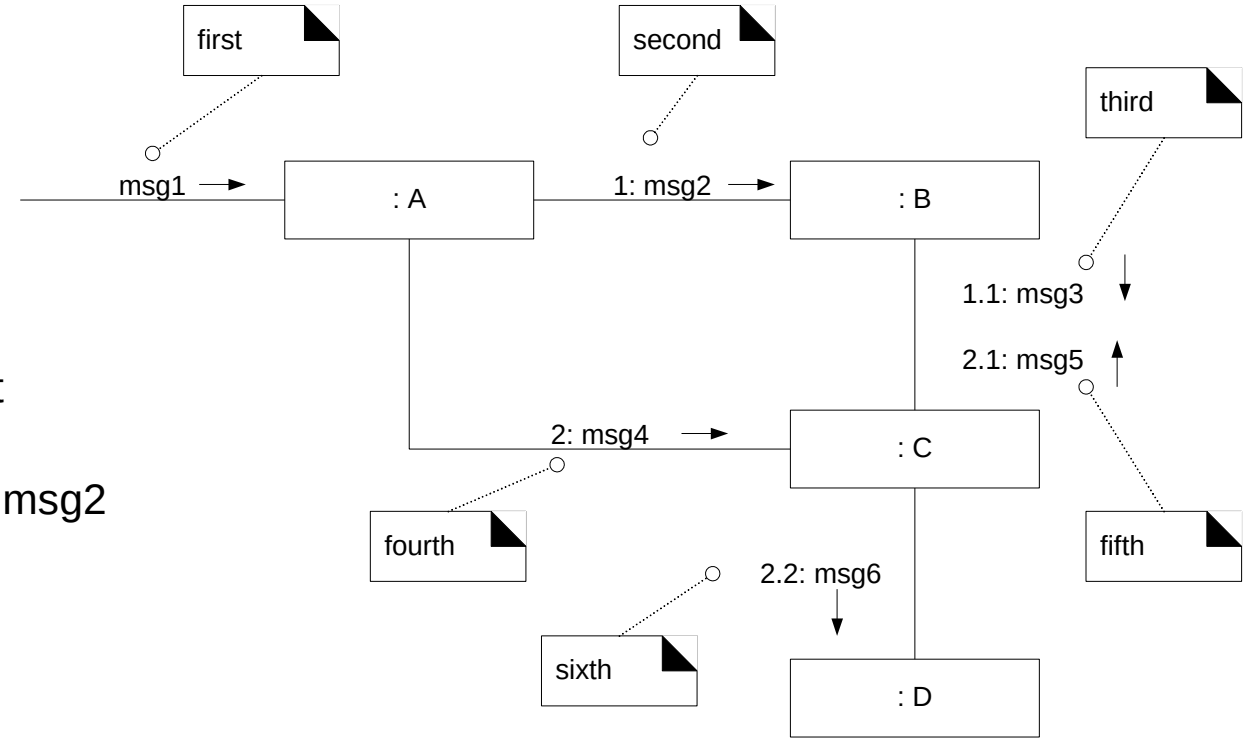
UML-Kommunikationsdiagramm (2/3)

- Notationselemente:
  - Lebenslinie (Box)
  - Synchrone Nachricht
    (= Aufruf einer Operation)
  - Antwortnachricht
    (= Rückgabewert)
  - Bedingte Nachrichten «[]»
  - Iteration «*»



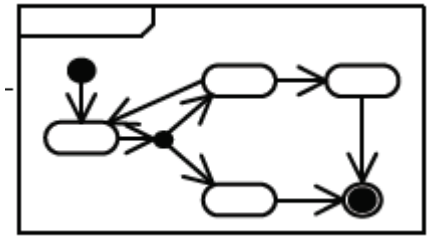
UML-Kommunikationsdiagramm (3/3)

- Nummerierung der Nachrichten
  - Wichtig für Festlegung der zeitlichen Abfolge!
  - Systemoperation (msg1, ohne Nummer)
  - Nummern gleicher Hierarchie (1.,2.,3., ...)
    - Operationen werden nacheinander aufgerufen
  - Nummern unterer Hierarchie
    - Werden innerhalb der Operation darüber ausgeführt (verschachtelt)
    - Operation 1.1:msg3 wird innerhalb von Operation 1:msg2 aufgerufen



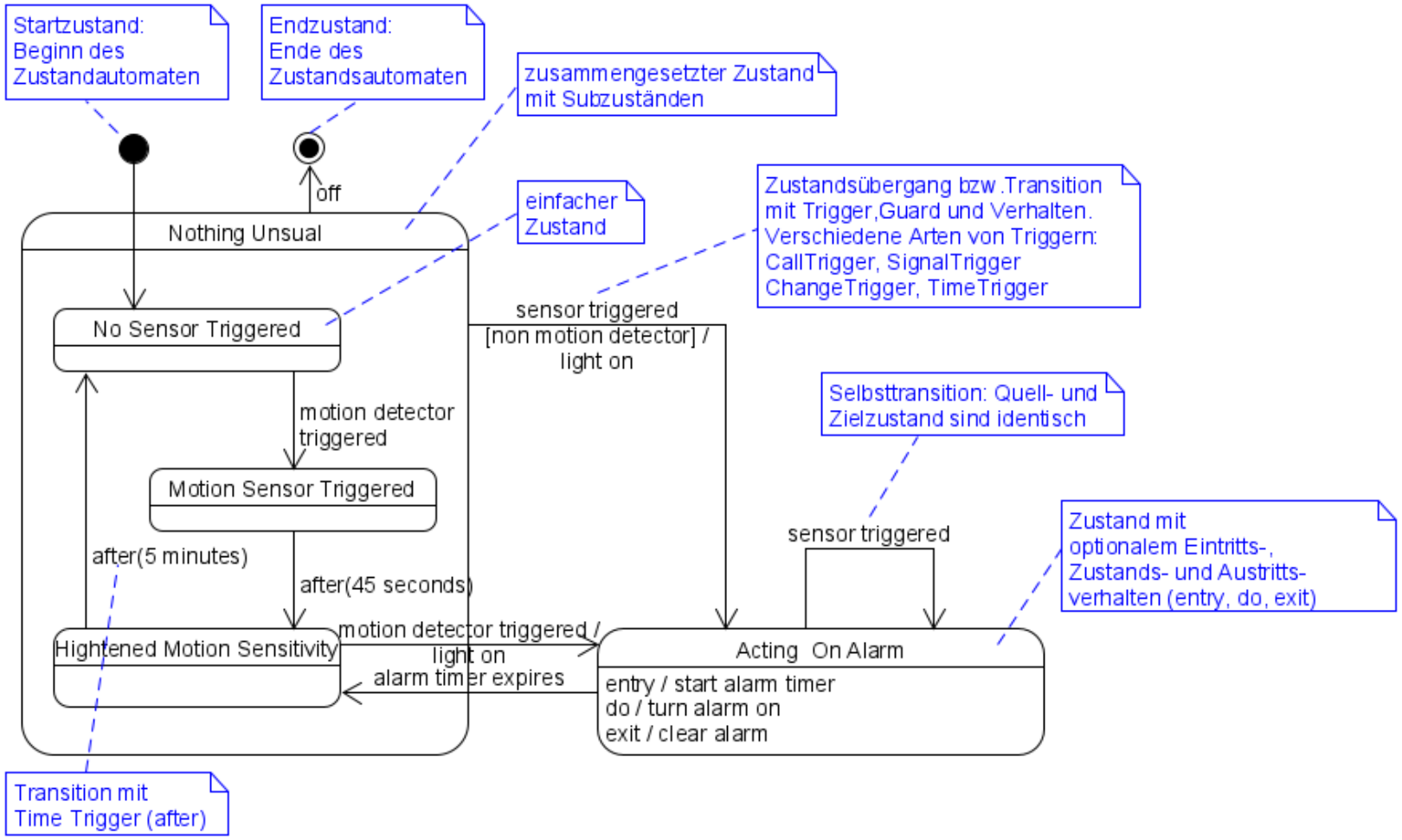
UML-Zustandsdiagramm (1/4)

- Das Diagramm beantwortet die zentrale Frage:
  Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, … bei welchen Ereignissen annehmen?
- Präzise Abbildung eines Zustandsmodells (endlicher Automat) mit Zuständen, Ereignissen, Nebenläufigkeiten, Bedingungen, Ein- und Austrittsaktionen.
- Zustände können wieder aus Zuständen bestehen (Schachtelung möglich).
- Das Zustandsdiagramm wird vor allem in der Modellierung von Echtzeitsystemen, Steuerungen und Protokollen verwendet.



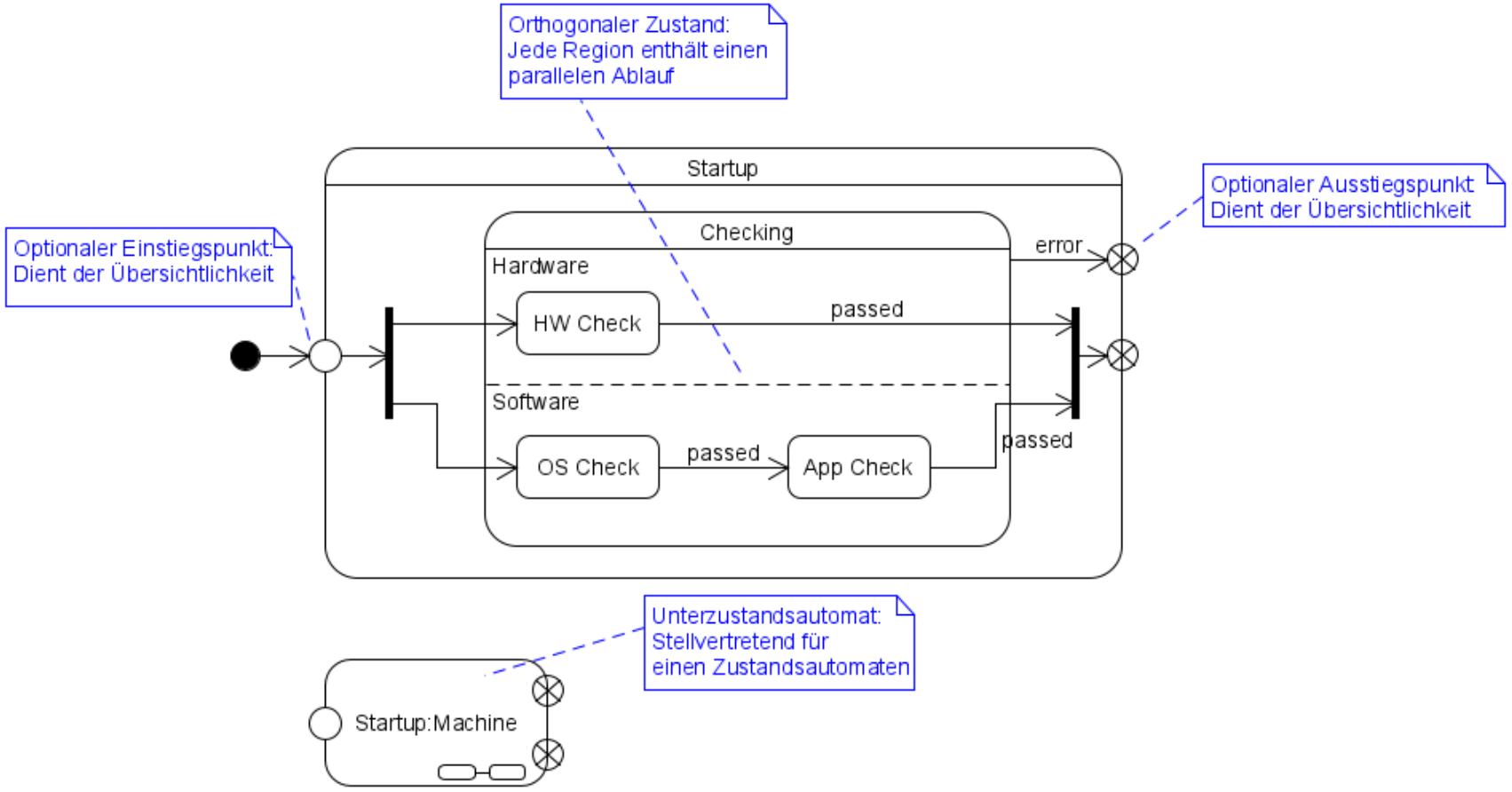
UML-Zustandsdiagramm (2/4)

- Notationselemente:
  - Start-, Endzustand
  - einfacher Zustand
  - Zusammengesetzter bzw. geschachtelter Zustand
  - Transition



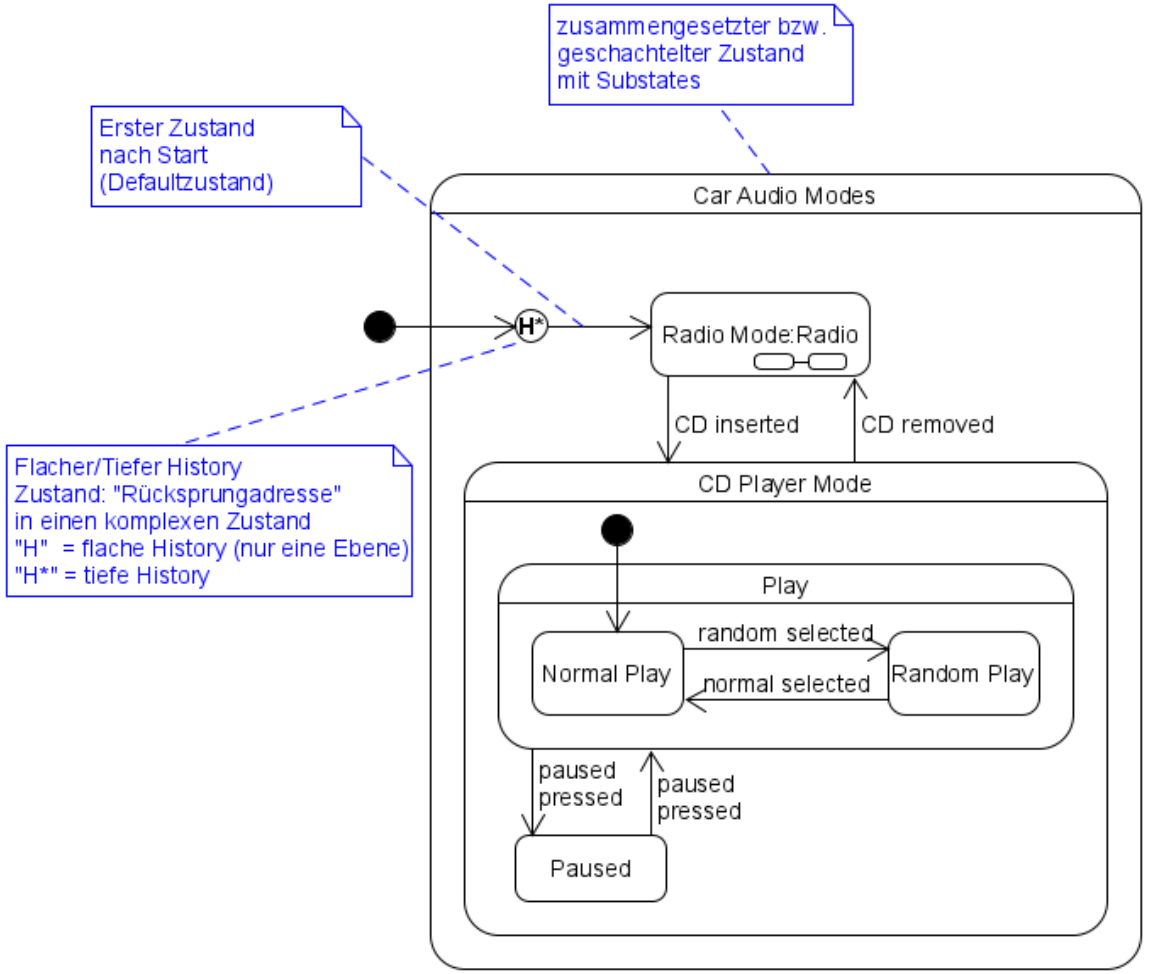
UML-Zustandsdiagramm (3/4)

- Notationselemente:
  - Orthogonaler Zustand
  - Parallelisierungsknoten
  - Synchronisationsknoten
  - Einstiegpunkt
  - Ausstiegspunkt
  - Unterzustandsautomat



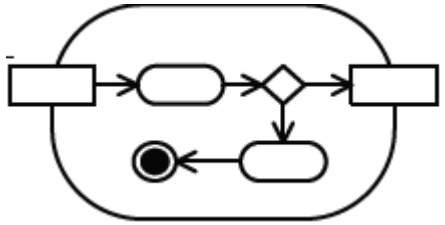
UML-Zustandsdiagramm (4/4)

- Notationselemente:
  - Zusammengesetzter Zustand
  - Flache und tiefe Historie



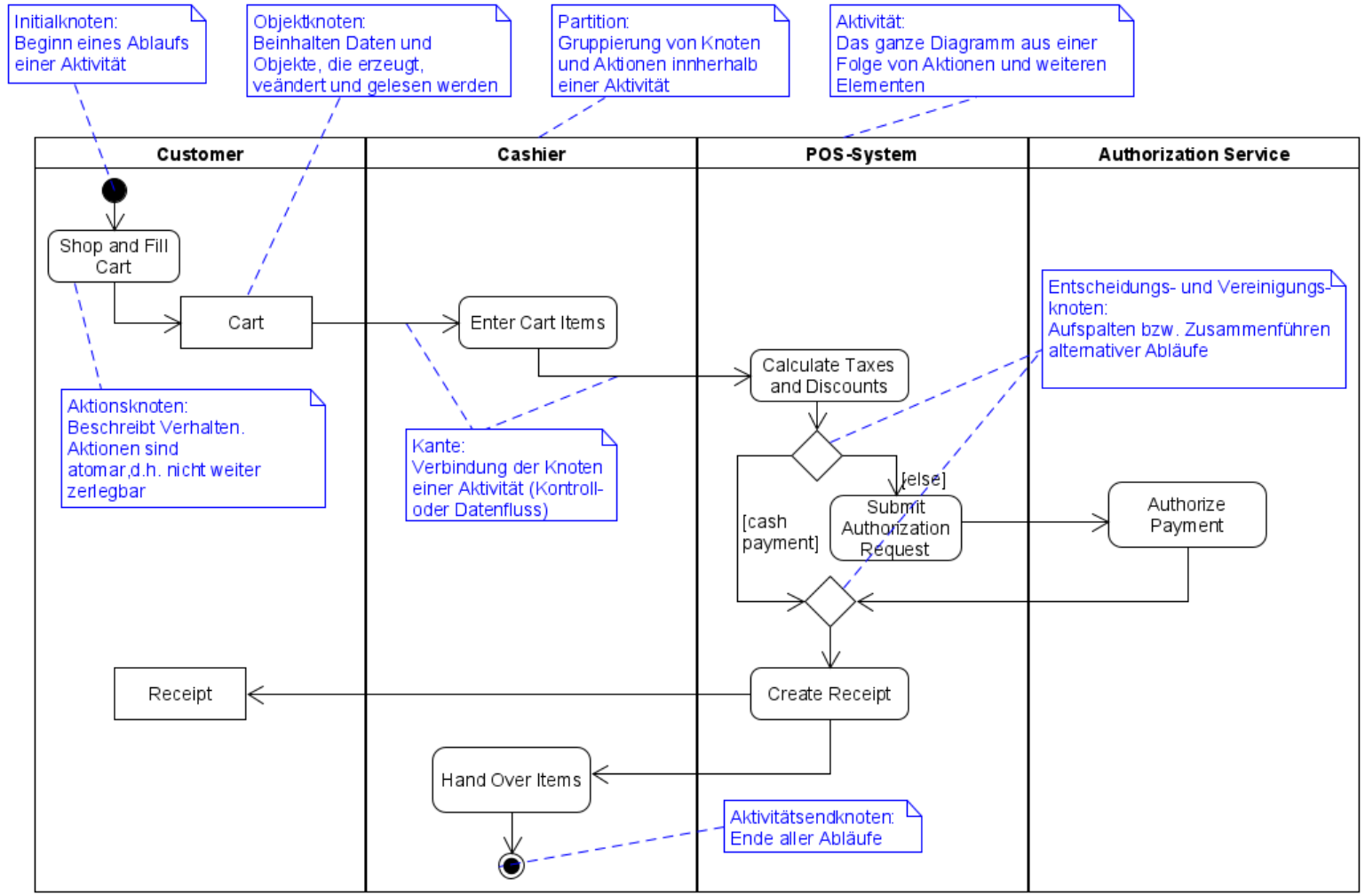
UML-Aktivitätsdiagramm (1/3)

- Das Diagramm beantwortet die zentrale Frage:
  Wie läuft ein bestimmter Prozess oder ein Algorithmus ab?
- Es kann eine sehr detaillierte Visualisierung von Abläufen mit Bedingungen, Schleifen und Verzweigungen modelliert werden.
- Es sind Parallelisierung und Synchronisation von Aktionen möglich.



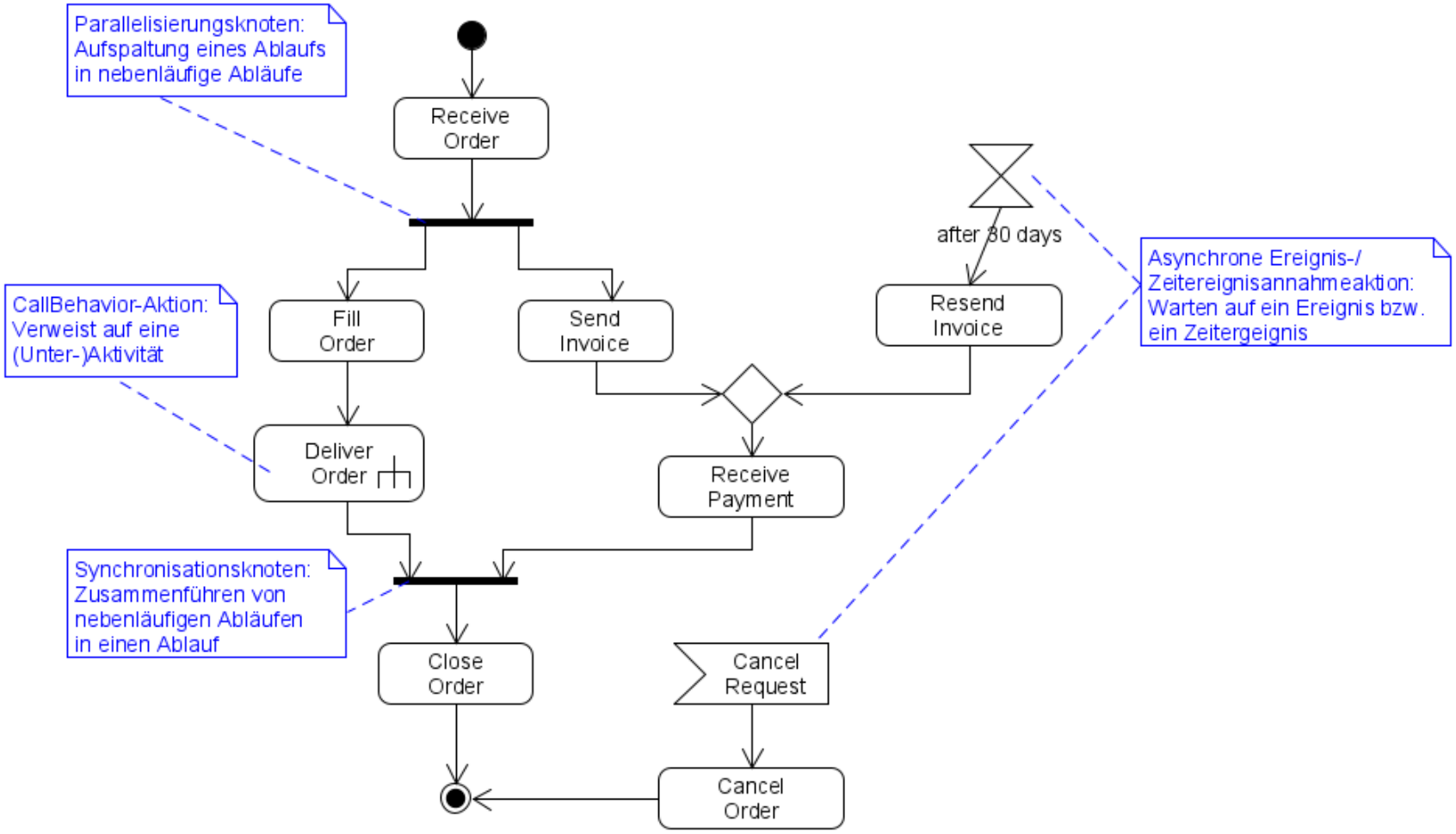
UML-Aktivitätsdiagramm (2/3)

- Notationselemente:
  - Aktivität
  - Aktionsknoten (Aktion)
  - Objektknoten (Objekt)
  - Entscheidungs- und Vereinigungsknoten
  - Kante
  - Initialknoten
  - Aktivitätsendknoten
  - Partition (auch Swimlane genannt)



UML-Aktivitätsdiagramm (3/3)

- Notationselemente:
  - Parallelisierungsknoten
  - Synchronisationsknoten
  - SendSignal-Aktion
  - Ereignis- bzw. Zeitereignisannahmeaktion
  - CallBehavior-Aktion



Weitere Informationen zur Modellierung mit der UML

- Diese Einführung in die Modellierung mit der UML und in die verschiedenen Diagramme für das Design ist eine kompakte Zusammenfassung, ohne detaillierte Erläuterung der Semantik.
- Sie umfasst der wichtigsten Notationselemente, mit denen ca. 80% der Problemstellungen modelliert werden können.

- *Achtung: Um damit in der Praxis modellieren zu können, müssen die Diagramme und Notationselemente in verschiedenen Problemstellungen angewendet und eingeübt werden!*

Tipps zur Modellierung

- Modellieren-im-Team: Skizzieren Sie erste Versionen im Team am Whiteboard. Verzichten Sie (vorläufig) auf komplexe Grafik- oder UML-Werkzeuge.
- Gerade-gut-genug: Vermeiden Sie das Streben nach Vollständigkeit Ihrer Modelle – meistens genügen Ausschnitte oder Teile des Systems für das Verständnis!
- Gerade-rechtzeitig: Verzögern Sie die Erstellung der ausgelieferten Dokumentation – es könnten sich ja noch Dinge ändern!
- Dokumentieren Sie kontinuierlich. Am Ende der letzten Iteration oder des Projekts können Sie sich nicht mehr an alle relevanten Dinge erinnern.
- Diagramm-plus-Text: Ergänzen Sie grafische Modelle um kurze textuelle Erläuterungen. Gute technische Dokumentation kombiniert Bild mit Text!
- Halten Sie Modelle möglichst redundanzfrei. Versuchen Sie, benötigte Dokumente aus einer einheitlichen Informations- oder Modellbasis zu generieren.

Denkpause

Aufgabe 6.1 (5’)

Szenario:

In einem Onlineshop kann ein Kunde Produkte bestellen. Eine Bestellung wird erst ausgeliefert, wenn der Kunde bezahlt hat. Storniert werden kann eine Bestellung bis zur Auslieferung. Eine abgewickelte Bestellung muss 10 Jahre lang archiviert werden. Daneben gelten noch weitere Regeln für die Abwicklung einer Bestellung

Diskutieren Sie in Murmelgruppen folgende Fragen:

- Mit was für dynamischen Modellen bzw. UML-Diagrammen kann dieses Szenario einer Bestellung in einem Onlineshop genauer modelliert und entworfen werden?

Agenda

1. Einführung in das objektorientierte Design
2. UML-Diagramme für das Design
3. **Klassen mit Verantwortlichkeiten entwerfen**
4. Wrap-up und Ausblick

Verantwortlichkeiten und Responsibility-Driven-Design

- Denken in Verantwortlichkeiten, Rollen und Kollaborationsbeziehungen für den Entwurf von Softwareklassen.
- Dieser Ansatz ist das Responsibility-Driven-Design (RDD).
- Softwareobjekte werden ähnlich wie Personen betrachtet, mit Verantwortlichkeiten und einer Zusammenarbeit mit anderen Personen, um eine Aufgabe zu erledigen.
- Verantwortlichkeiten werden durch Attribute und Methoden implementiert.
  - Evtl. in Zusammenarbeit mit Operationen von anderen Klassen bzw. Objekten.
- RDD kann auf jeder Ebene des Designs angewendet werden (Klasse, Komponente, Schicht).

Ausprägungen von Verantwortlichkeiten

- «Doing»-Verantwortlichkeiten (oder Algorithmen, Code)
  - Selbst etwas tun
  - Aktionen anderer Objekte anstossen
  - Aktivitäten anderer Objekte kontrollieren und steuern

- «Knowing»-Verantwortlichkeit (oder Daten, Attribute)
  - Private eingekapselte Daten
  - Verwandte Objekte kennen
  - Dinge kennen, die es ableiten oder berechnen kann
  - Daten/Objekte zur Verfügung stellen, die aus den bekannten Daten/Objekten abgeleitet oder berechnet werden können

GRASP: Ein methodischer Ansatz für das OO-Design

- GRASP (General Responsibility Assignment Software Patterns) bezeichnet eine Menge von grundlegenden Prinzipen bzw. Pattern, mit denen die Zuständigkeit bestimmter Klassen objektorientierter Systeme festgelegt wird.
- Sie beschreiben allgemein welche Klassen und Objekte wofür zuständig sein sollten (Verantwortlichkeiten und Kollaborationen).
- Diese allgemein anerkannten Regeln wurden von Craig Larman [1] systematisch beschrieben.
- Dies erleichtert die Kommunikation zwischen Softwareentwicklern und erleichtert Einsteigern als Lernhilfe das Entwickeln eines Bewusstseins für guten bzw. schlechten Code.

Prinzipien und Pattern

- Ein Prinzip ist in diesem Kontext ein Grundsatz oder Postulat, das zu einem guten objektorientierten Design führen soll.

- Ein Pattern ist ein benanntes Problem-Lösungspaar, das in neuen Kontexten angewendet werden kann.

- Weitere Design-Patterns siehe Gang-of-Four (GoF)
  - Oft wiederkehrende Problemstellungen mit detaillierten Lösungsmustern

GRASP – Neun Prinzipien bzw. Patterns

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

GRASP – Neun Prinzipien bzw. Patterns

- Jeder Teilnehmende holt sich einen Zettel aus der Themenbox
- Erarbeiten Sie das/die Pattern/Prinzipien selbständig. Recherchieren Sie!
  - Wozu wird das Pattern/Prinzip verwendet?
  - Wie erkenne ich das Pattern/Prinzip?
  - Wie wende ich das Pattern/Prinzip an?
  - Zeigen Sie das Pattern/Prinzip in einem UML-Diagramm

- Alle Experten zu einem Thema finden sich zusammen, sichten die einzelnen Lösungsteile und erarbeiten eine gemeinsame Aussage, die vorgetragen werden kann.
- Ein Experte wird bestimmt, das Pattern/Prinzip allen vorzutragen und erklären.

GRASP – Neun Prinzipien bzw. Patterns

- Bitte die Klasse durchnummerieren und Nummer merken
- Bilden Sie den Modulo 9 mit Ihrer Nummer
- Erarbeiten Sie das/die Pattern/Prinzipien selbständig. Recherchieren Sie!
  - Wozu wird das Pattern/Prinzip verwendet?
  - Wie erkenne ich das Pattern/Prinzip?
  - Wie wende ich das Pattern/Prinzip an?
  - Zeigen Sie das Pattern/Prinzip in einem UML-Diagramm

GRASP – Neun Prinzipien bzw. Patterns

- Information Expert (0)
- Creator (1)
- Controller (2)
- Low Coupling (3)
- High Cohesion (4)
- Polymorphism (5)
- Pure Fabrication (6)
- Indirection (7)
- Protected Variations (8)

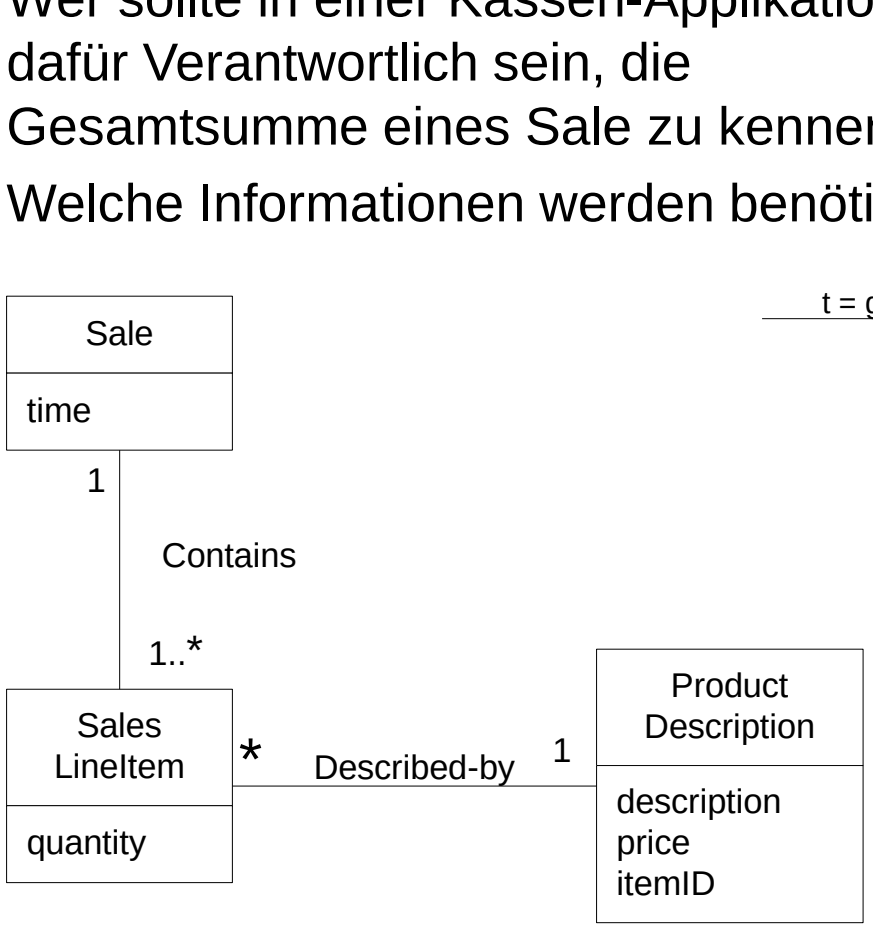
- Alle Experten zu einem Thema finden sich zusammen, sichten die einzelnen Lösungsteile und erarbeiten eine gemeinsame Aussage, die vorgetragen werden kann.
- Ein Experte wird bestimmt, das Pattern/Prinzip allen vorzutragen und zu erklären.

Information Expert (1/2)

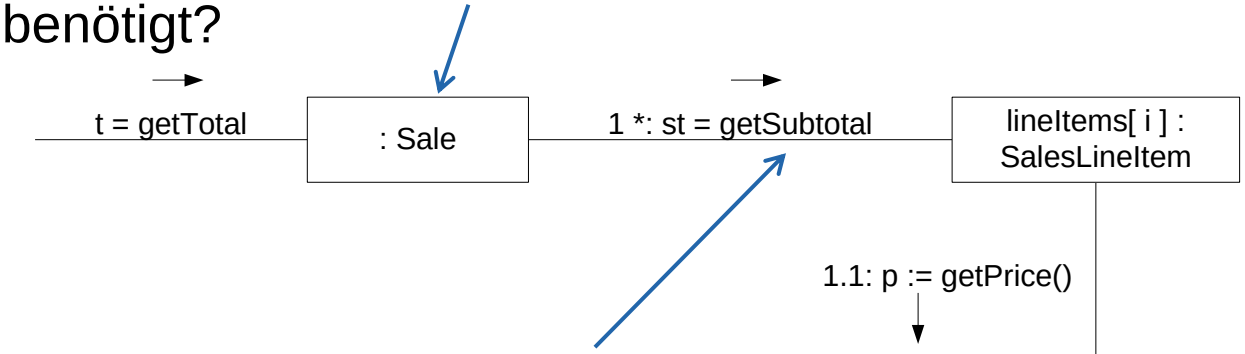
- **Name:** Information Expert
- **Problem:** Gibt es ein grundlegendes Prinzip, um Objekten Verantwortlichkeiten zuzuweisen?
- **Lösung, Ratschläge**: Weisen Sie die Verantwortlichkeit einer Klasse zu, die über die erforderlichen Informationen verfügt, um sie zu erfüllen.
- Alternativen: Low Coupling oder High Cohesion erfordern andere Lösung, nämlich eine «künstliche» Klasse.
- Es sind auch partielle Verantwortlichkeiten möglich.

Information Expert (2/2)

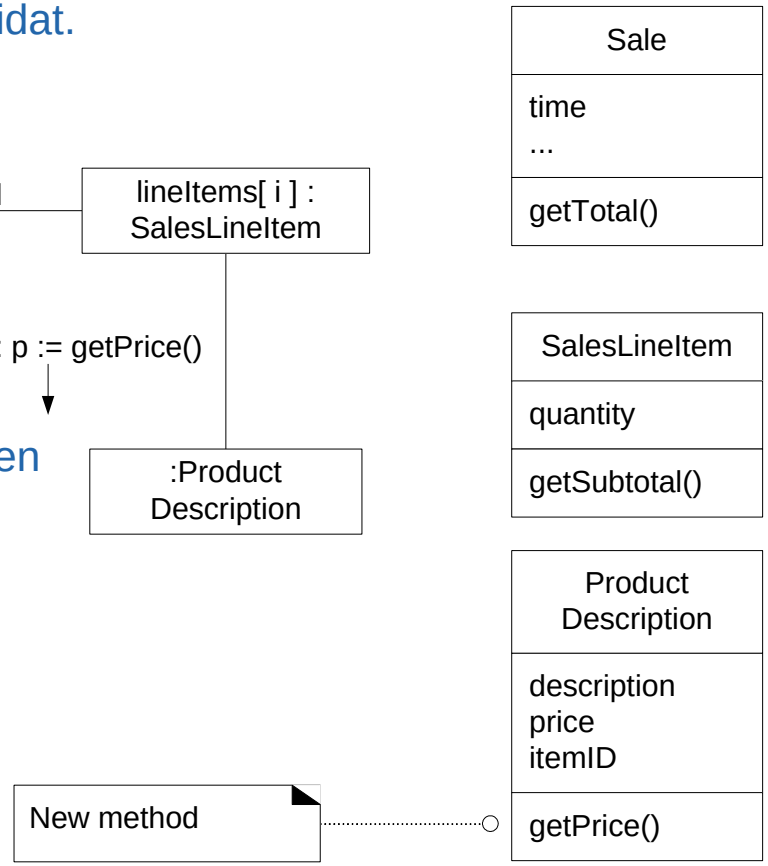
- Wer sollte in einer Kassen-Applikation dafür Verantwortlich sein, die Gesamtsumme eines Sale zu kennen?
- Welche Informationen werden benötigt?



Ein Sale-Objekt kennt alle seine SalesLineItem und ist darum laut Information Expert ein Kandidat.



Ein SalesLineItem-Objekt kennt den Preis seines Produkts und soll darum laut Information Expert die Zwischensumme bestimmen.



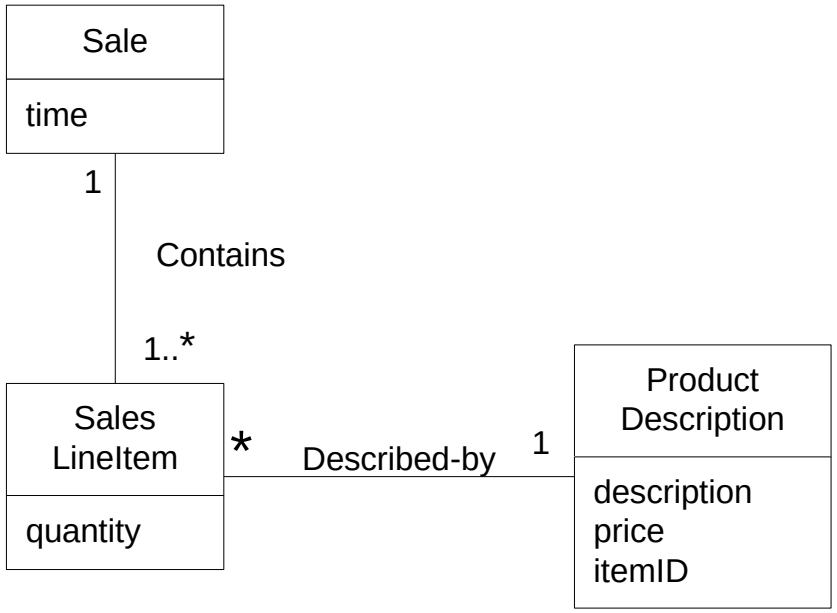
Creator (1/2)

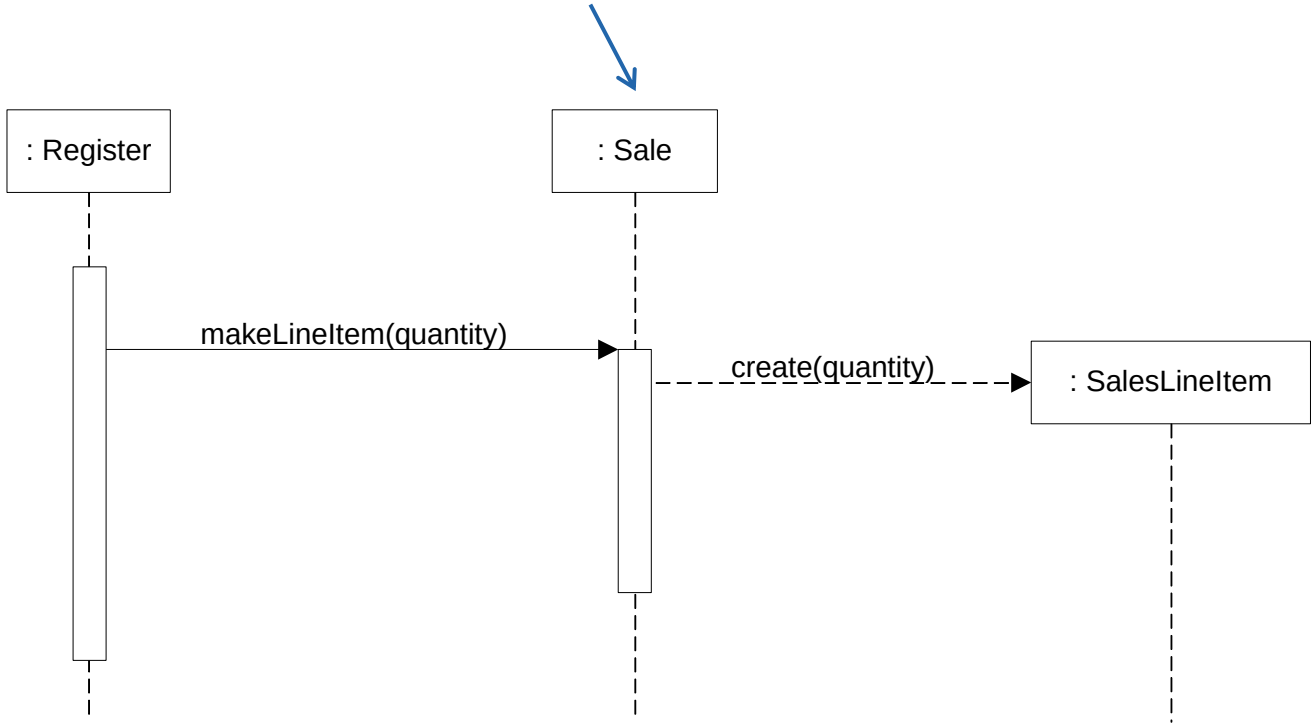
- **Name:** Creator
- **Problem:** Wer soll dafür Verantwortlich sein, eine neue Instanz (Objekt) einer Klasse zu erzeugen?
- **Lösung, Ratschläge**: Weisen Sie einer Klasse A die Verantwortlichkeit zu, eine Instanz der Klasse B zu erstellen, wenn eine oder mehrere der folgenden Aussagen wahr ist/sind (je mehr desto besser):
  - A eine Aggregation oder ein Kompositum von B ist
  - A registriert oder erfasst B-Objekte
  - A arbeitet eng mit B-Objekten zusammen oder hat ein enge Kopplung
  - A verfügt über Initialisierungsdaten für B (d.h. A ist Experte bezüglich Erzeugung von B)
- Wenn mehrere Optionen anwendbar sind, sollten Sie eine Klasse A vorziehen, die ein Aggregat oder ein Kompositum ist.
- Alternativen: Factory Pattern, Dependency Injection (DI)

Creator (2/2)

- Wer sollte in einer Kassen-Applikation für die Erstellung eines SalesLineItem verantwortlich sein?

Da ein Sale-Objekt viele SalesLineItem enthält (Komposition), ist es laut Creator Pattern ein guter Kandidat





Controller (1/2)

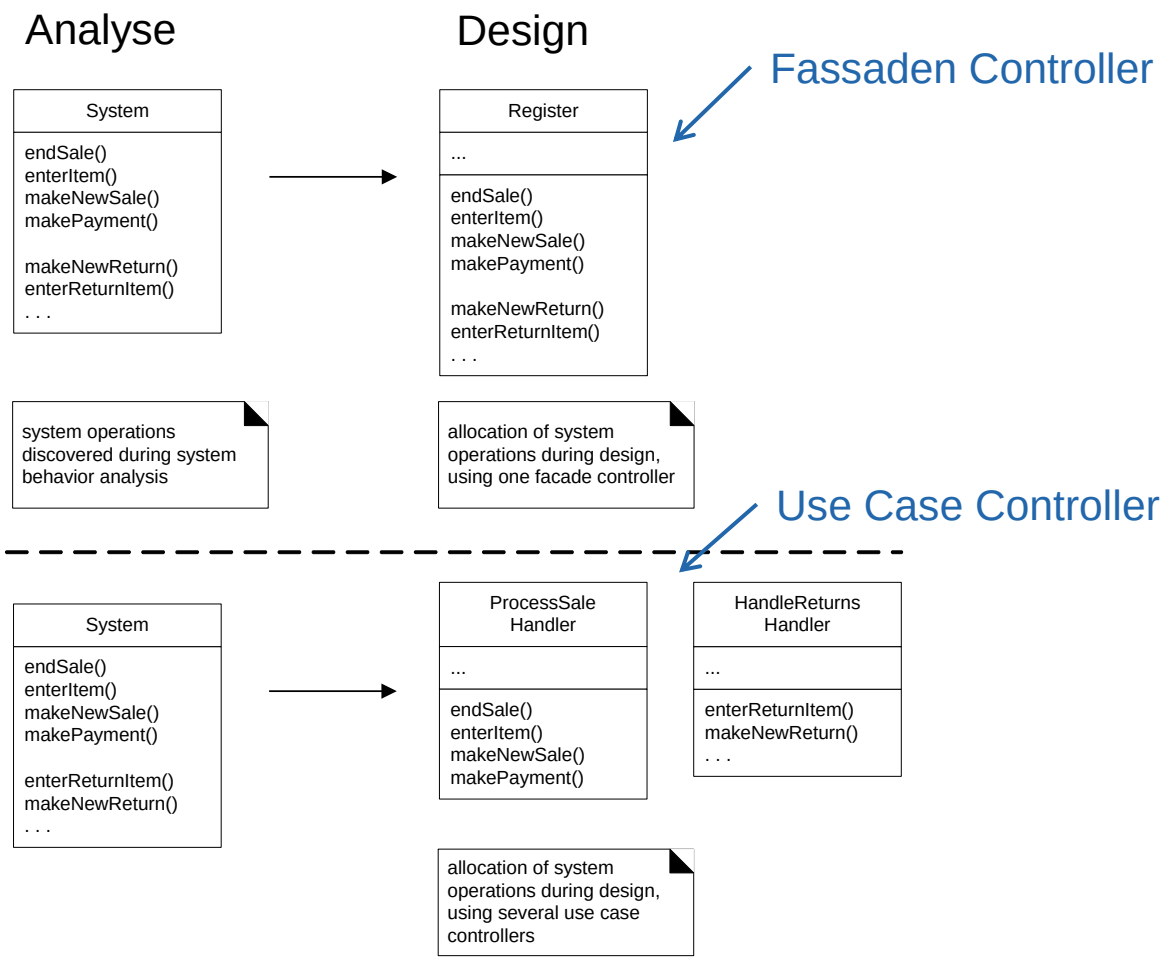
- **Name:** Controller
- **Problem:** Welches erste Objekt jenseits der UI-Schicht empfängt und koordiniert («kontrolliert») eine Systemoperation?
- Systemoperationen wurden zuerst in der Analyse von System-Sequenzdiagrammen eingeführt. Diese nehmen die Input-Ereignisse unsers Systems entgegen.
- **Lösung, Ratschläge**: Weisen Sie Verantwortlichkeit einer Klasse zu, die eine der folgenden Bedingungen erfüllt:
  - Variante 1: Fassaden Controller
    Sie repräsentiert das «Root-Objekt», System bzw. übergeordnetes System auf dem die Software läuft.
  - Variante 2: Use Case Controller
    Pro Use-Case-Szenario eine «künstliche» Klasse, in der die Systemoperation abläuft.
- Wichtig: Controller macht selber nur wenig und delegiert fast alles!

Controller (2/2)

- Welche Klasse sollte der Controller für die Systemoperationen einer Kassen-Applikation sein?

System
endSale() enterItem() makeNewSale() makePayment() . . .

Wenn ein Fassaden Controller eine zu geringe Kohäsion aufweist und zu gross wird (LOC), ist ein Use Case Controller zu präferieren!



Low Coupling (1/2)

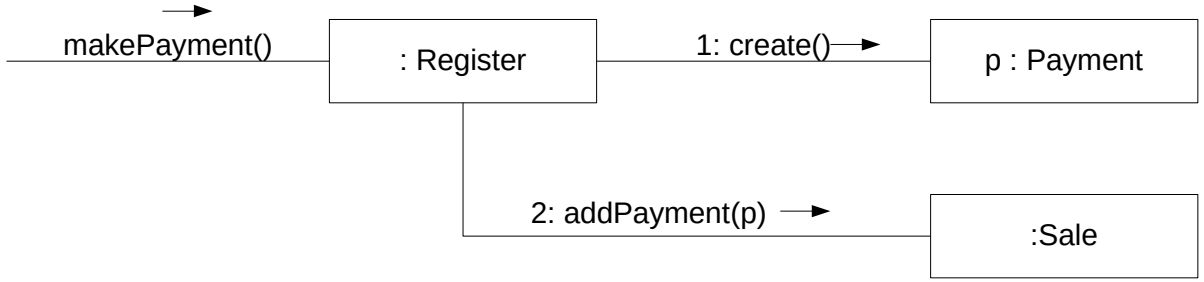
- **Name:** Low Coupling
- **Problem:** Wie erreicht man eine geringe Abhängigkeit, wie begrenzt man die Auswirkungen von Änderungen und wie verbessert man die Wiederverwendbarkeit?
  - Kopplung = Mass für die Abhängigkeit von anderen Elementen (Klassen, Subsysteme, Systeme)
  - Hohe Kopplung: Element ist von vielen anderen Elementen abhängig.
  - Niedrige Kopplung: Element ist nur von wenigen anderen Elementen abhängig.
  - Klassen mit hoher Kopplung leiden häufig unter folgenden Problemen:
    - aufgrund von Änderungen in verbundenen Klassen sind oft lokale Änderungen nötig
    - schwieriger zu verstehen
    - schwieriger wiederzuverwenden, weil für ihre Verwendung auch die Klassen vorhanden sein müssen, von denen sie abhängig sind
- **Lösung, Ratschläge**:
  - Weisen Sie Verantwortlichkeiten so zu, dass die Kopplung gering bleibt.
  - Bewerten Sie anhand dieses Prinzips mögliche Alternativen (vgl. das I in SOLID [3]).

Low Coupling (2/2)

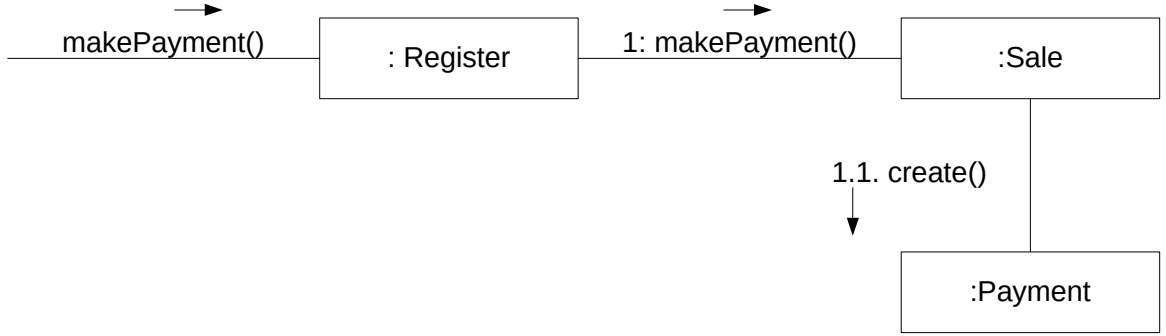
- Wer sollte in einer Kassen-Applikation Payment erzeugen?

Low Coupling präferiert das Design 2. Dadurch ist eine Kopplung weniger vorhanden.

Design 1



Design 2



Anmerkung: Payment ist in beiden Varianten mit Sale verbunden (Assoziation im DCD).

High Cohesion (1/2)

- **Name:** High Cohesion
- **Problem:** Wie kann erreicht werden, dass Objekte fokussiert, verständlich und handhabbar bleiben und nebenbei Low Coupling unterstützen?
  - Kohäsion (oder spezieller funktionale Kohäsion) ist ein Mass für die Verwandtschaft und Fokussierung eines Elements (Elemente können Klassen, Subsysteme usw. sein).
  - Hohe Kohäsion: Element erledigt nur wenige Aufgaben, die eng miteinander verwandt sind
  - Geringe Kohäsion: Element, das für viele unzusammenhängende Dinge verantwortlich ist
  - Klassen mit niedriger Kohäsion leiden häufig unter den folgenden Problemen:
    - schwierig zu verstehen
    - schwierig wiederzuverwenden
    - brüchig und instabil, sind laufend von Änderungen betroffen
- **Lösung, Ratschläge**:
  - Weisen Sie Verantwortlichkeiten so zu, dass die Kohäsion hoch bleibt.
  - Verwenden Sie dieses Kriterium, um Alternativen zu bewerten (vgl. S in SOLID [3]).

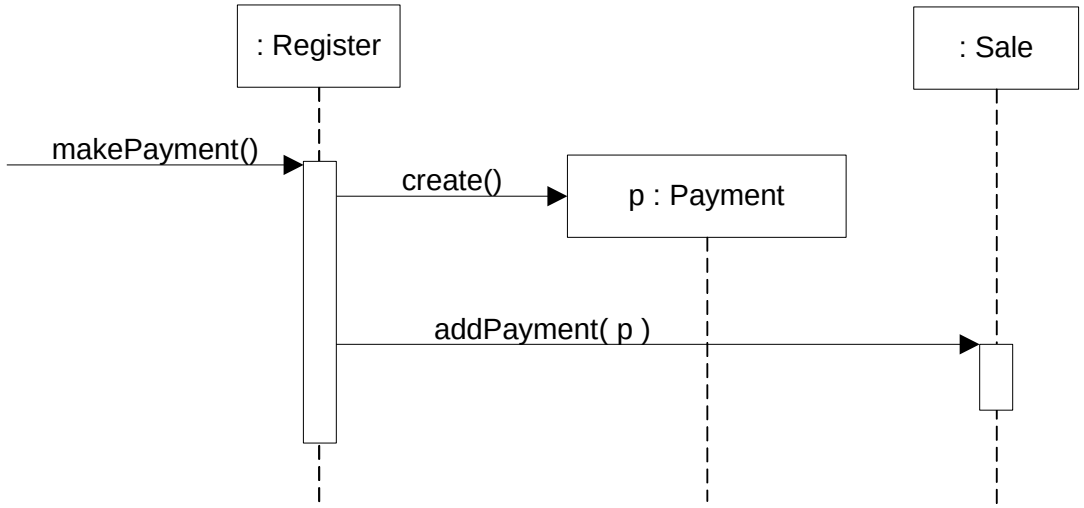
High Cohesion (2/2)

- Wie ist das Beispiel aus Low Coupling in Bezug auf High Cohesion zu bewerten?

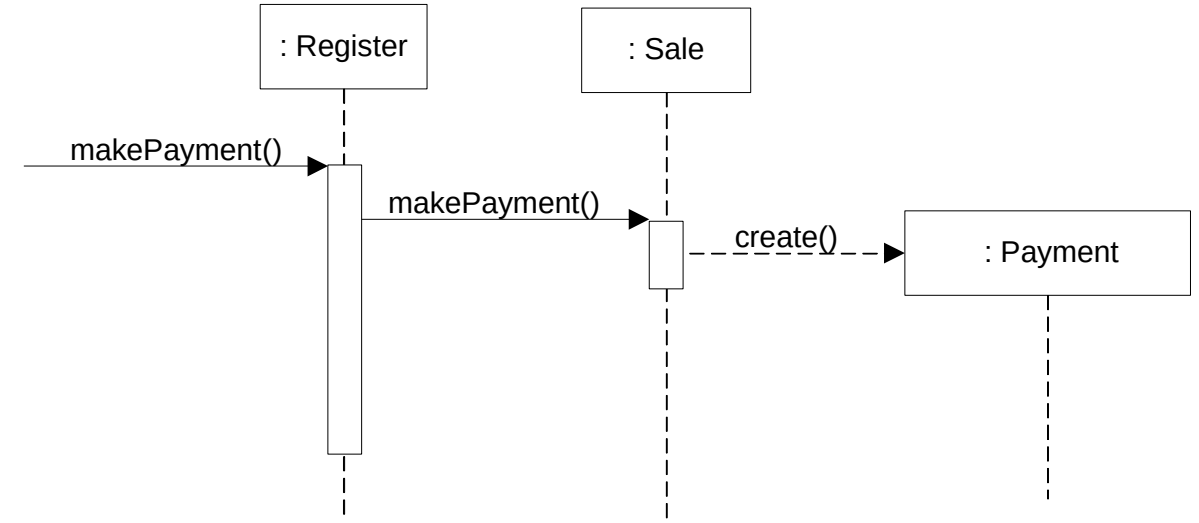
Das Design 2 ist zu präferieren, da es eine hohe Kohäsion und eine geringere Kopplung aufweist.

Kohäsion und Kopplung sind das Yin und Yang des Software Engineering, weil sie sich gegenseitig beeinflussen!

Design 1



↘ Design 2

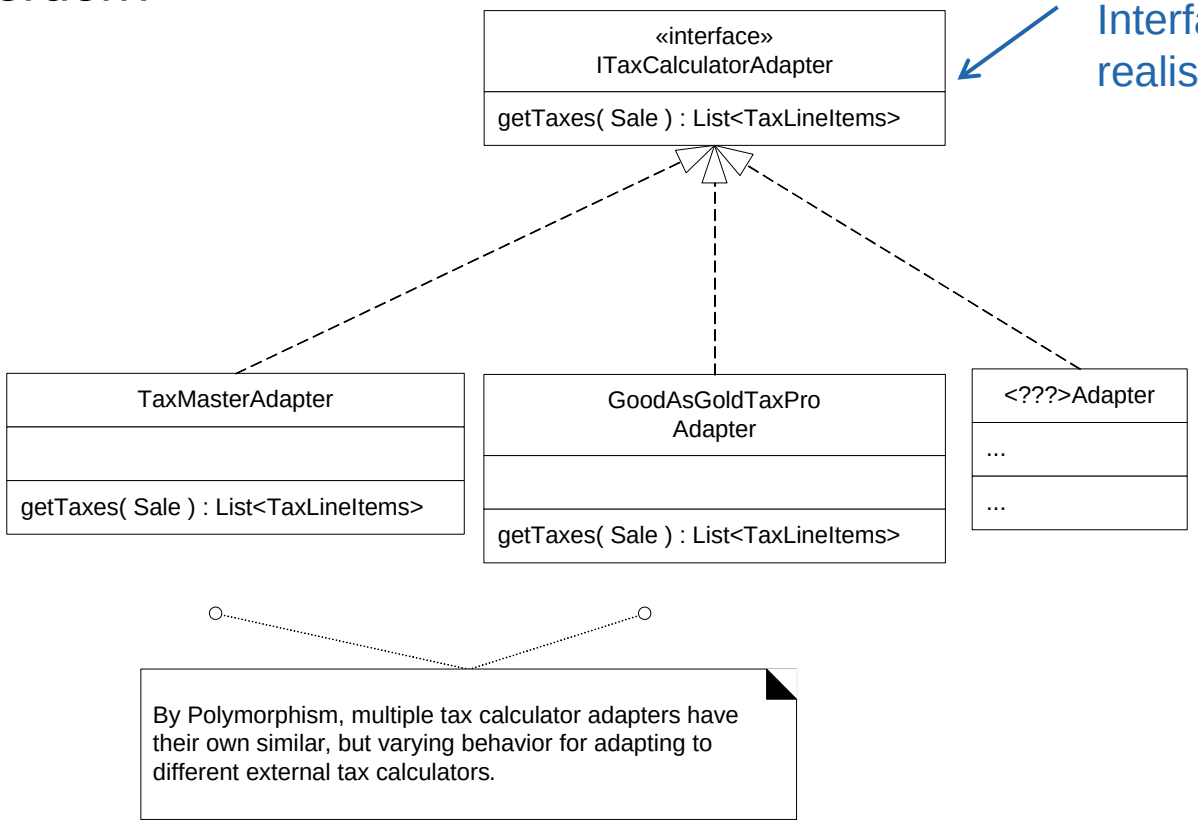


Polymorphism (1/2)

- **Name:** Polymorphism
- **Problem:** Wie werden typabhängige Alternativen gehandhabt?
  - Operation weist viele if-then-else bzw. grosse switch-case Anweisungen auf
  - Sie möchten ein bestimmtes Verhalten (z.B. Einsatz eines externen Dienstes) konfigurierbar machen.
- **Lösung, Ratschläge**: Weisen Sie das typabhängige Verhalten mit polymorphen Operationen der Klasse zu, dessen Verhalten variiert.
  - Dies ist eine der grundlegenden Ideen in der objektorientierten Programmierung (Generalisierung / Spezialisierung).
  - Achtung: Überprüfen Sie, ob es sich tatsächlich auch um eine «is a» Beziehung zwischen Superklasse und Subklassen handelt.
  - Dabei sollte auch das sogenannte Liskov-Substitutions-Prinzip (vgl. L in SOLID [3]) eingehalten werden (s. Wikipedia, abgerufen am 30.6.2020).

Polymorphism (2/2)

- Wie soll das Steuerberechnungsmodul in einer Kassen-Applikation von Drittanbietern eingebunden werden?



Polymorphism kann mit einer konkreten Klasse, einer abstrakten Klasse, Interfaces oder einer Kombination realisiert werden.

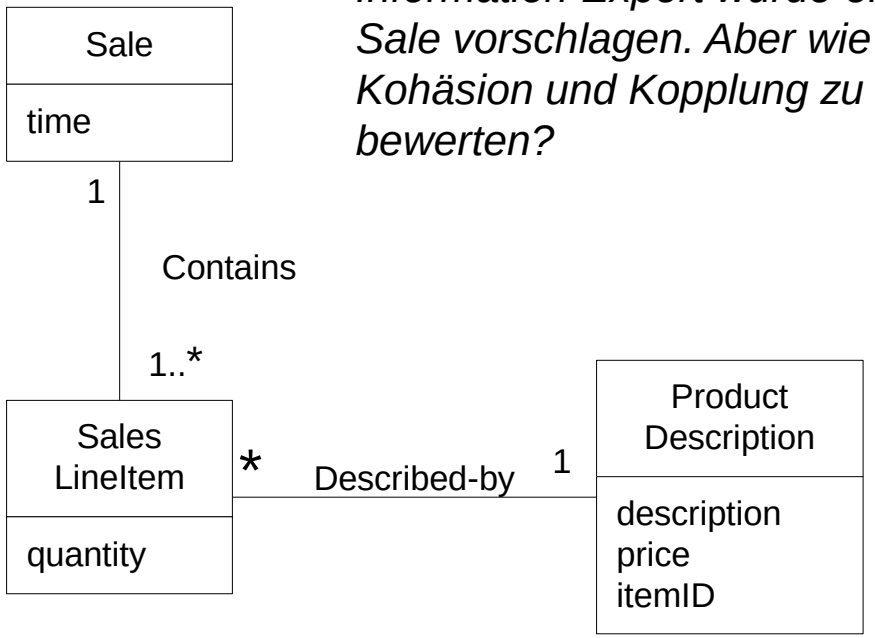
Pure Fabrication (1/2)

- **Name:** Pure Fabrication
- **Problem:** Welches Objekt sollte die Verantwortlichkeit haben, wenn Sie nicht gegen High Cohesion und Low Coupling oder andere Ziele verstossen wollen, aber die Lösungen, die beispielsweise vom Information Expert vorgeschlagen werden, nicht passen?
  - Viele Design-Klassen können direkt aus dem Fachbereich (Domänenmodell) abgeleitet werden und erfüllen das Low Representational Gap.
  - Aber es gibt auch viele Situationen, wo es Probleme mit einer geringen Kohäsion, einer starken Kopplung und einer geringen Wiederverwendung gibt, wenn die Verantwortlichkeiten der Klasse in der Domänenschicht zugewiesen wird (s. nächste Folie).

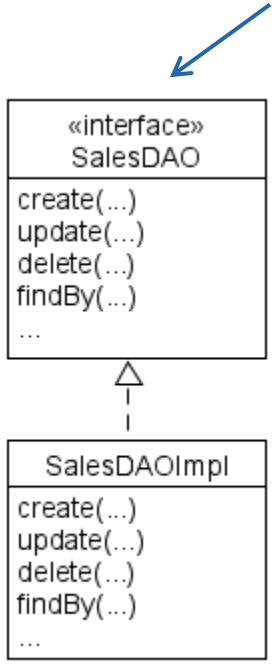
- **Lösung, Ratschläge**:
  - Weisen Sie einen hoch kohäsiven Satz von Verantwortlichkeiten einer künstlichen Hilfsklasse zu
  - Wird nur erstellt, um eine hohe Kohäsion, eine geringe Kopplung oder eine bessere Wiederverwendbarkeit zu realisieren.

Pure Fabrication (2/2)

- Wer soll Sales-Instanzen in einer Kassenapplikation in der Datenbank speichern?



Information Expert würde eigentlich Sale vorschlagen. Aber wie sind Kohäsion und Kopplung zu bewerten?



Der Aspekt der Persistenz wird nach Pure Fabrication in einer neuen «künstlichen» Klasse behandelt. (s. dazu später LE12 Vertiefung 3: Persistenz)

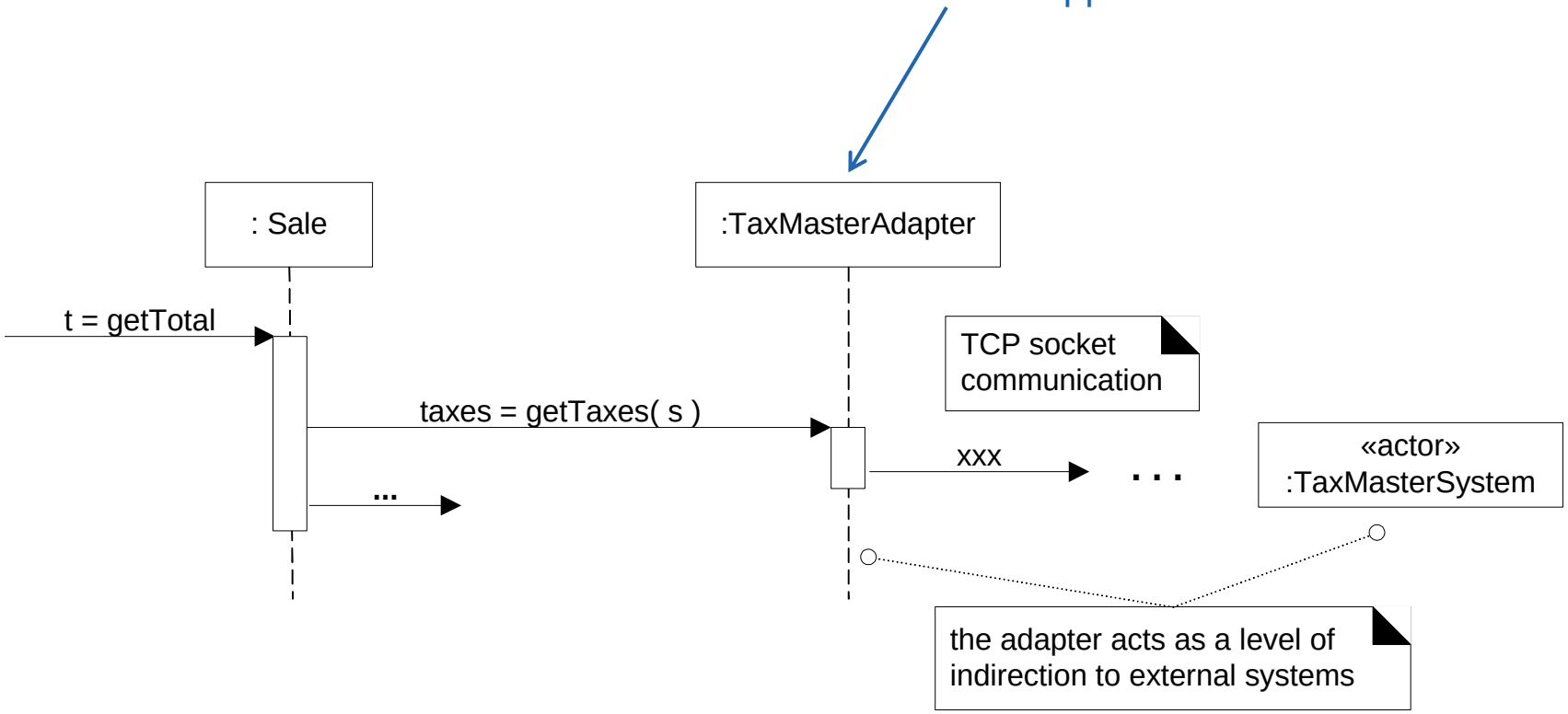
Indirection (1/2)

- **Name:** Indirection
- **Problem:** Wie soll eine Verantwortlichkeit zugewiesen werden, um eine direkte Kopplung zwischen zwei (oder mehr) Objekten zu vermeiden? Wie können Objekte entkoppelt werden, so dass die Kopplung geringer und das Wiederverwendungspotential grösser wird?
- **Lösung, Ratschläge**: Weisen Sie die Verantwortlichkeit einem zwischengeschalteten Objekt zu, das zwischen den anderen Komponenten oder Diensten vermittelt, so dass diese nicht direkt gekoppelt sind (vgl. das D in SOLID [3]).
- Der Vermittler schafft eine Indirektion zwischen den anderen Komponenten.
- Alternativen: Protected Variations
- Viele GoF Design Patterns wie Adapter, Bridge, Facade, Observer oder Mediator verwenden dieses Prinzip.
- Viele Indirections sind Pure Fabrications.

Indirection (2/2)

- Wie kann eine direkte Kopplung zwischen Sale und der Steuerberechnung vermieden werden?

Indirection und Polymorphism entkoppeln Sales von der externen API.



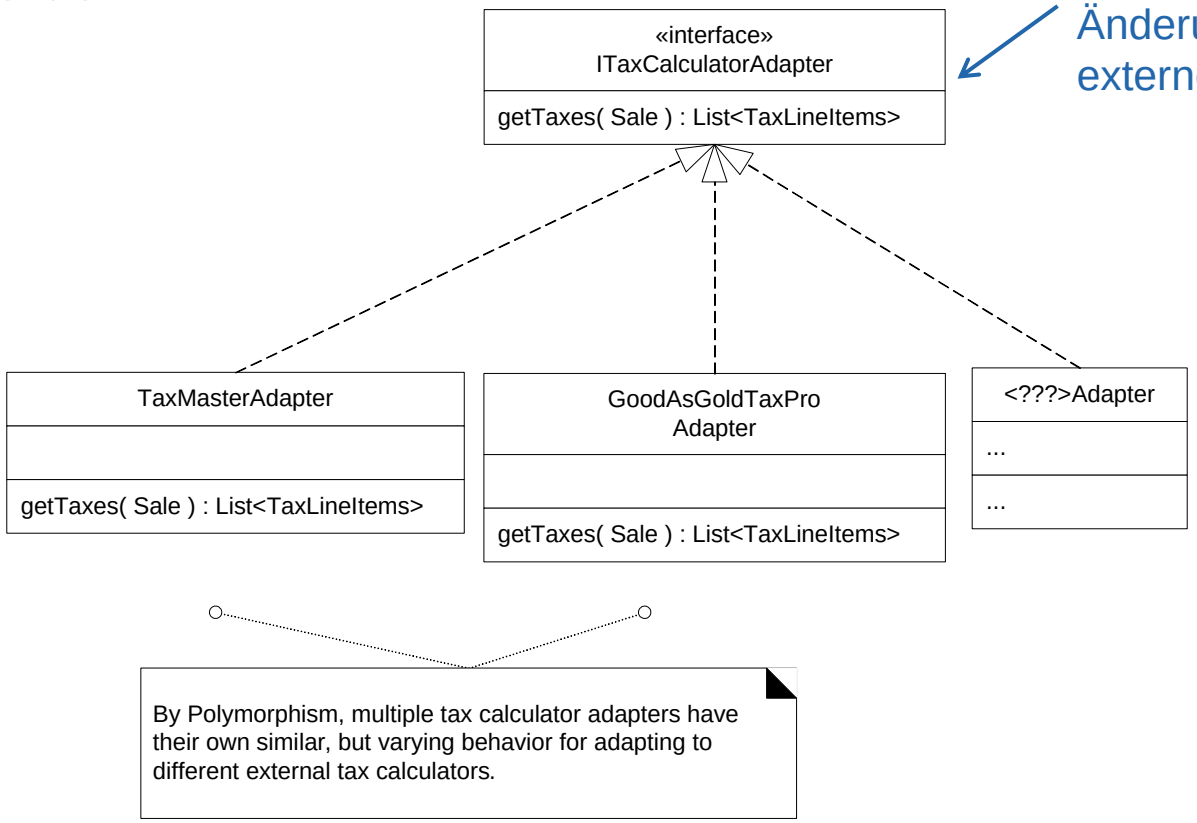
Protected Variations (1/2)

- **Name:** Protected Variations
- **Problem:** Wie sollen Objekte, Subsysteme und Systeme entworfen werden, sodass Veränderungen und Instabilitäten in diesen Elementen keinen Einfluss auf andere Elemente haben?
- **Lösung, Ratschläge**: Identifizieren Sie die Punkte, an denen Veränderungen und Instabilitäten zu erwarten sind; weisen Sie Verantwortlichkeiten so zu, dass diese Punkte durch ein stabiles Interface eingekapselt werden (vgl. das O und D in SOLID [3]).
- Dies ist ein sehr wichtiges, grundlegendes Prinzip des Softwaredesigns!
- Es sollte zwischen folgenden Änderungspunkten unterschieden werden.
  - Variationspunkt: Veränderungen sind sicher (in Anforderung); Zwingend PV Konzepte einbauen
  - Entwicklungspunkt: Veränderungen sind nicht sicher, werden aber mit hoher Wahrscheinlichkeit eintreffen; sind nicht in Anforderungen enthalten
- Spekulative Anwendungen sind aber zu vermeiden, da dies zu unnötiger Komplexität führt.
- Es ist die «Kunst» des erfahrenen Designers, die richtigen Annahmen treffen!

Protected Variations (2/2)

- Wie soll das Steuerberechnungsmodul in einer Kassen-Applikation von Drittanbietern eingebunden werden?

Dies ist auch ein Beispiel für Protected Variations. Der Punkt der Variabilität oder Änderungen sind die verschiedenen APIs der externen Steuerberechnung.



Agenda

1. Einführung in das objektorientierte Design
2. UML-Diagramme für das Design
3. Klassen mit Verantwortlichkeiten entwerfen
4. **Wrap-up und Ausblick**

Wrap-up

- Das wichtigste Ziel des objektorientierten Designs ist es, Klassen mit klaren Verantwortlichkeiten und Kollaborationen zu entwerfen.
- Eine Use-Case-Realisierung wird mit einem Design-Klassendiagramm (DCD) und mehreren Interaktionsdiagrammen modelliert, um das Design zu diskutieren und evaluieren zu können.
- Dabei werden das Design-Klassendiagramm und die weiteren Modellierungsartefakte schrittweise erweitert und ergänzt durch jedes entworfene und implementierte Use-Case-Szenario.
- GRASP ist eine Lernhilfe, um beim Design bewusst die Verantwortlichkeiten und Kollaborationen zwischen Objekten festzulegen.
- GRASP sind grundlegende Prinzipien und Patterns, die zu einem guten objektorientierten Design führen und das Design nachvollziehbar begründen.

Ausblick

- In der nächsten Lerneinheit werden wir:
  - wichtige Aspekte für die Implementation, Refactoring und Testing einer Use-Case-Realisierung diskutieren.

Quellenverzeichnis

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

[2] Seidel, M. et al.: UML @ Classroom: Eine Einführung in die objektorientierte Modellierung, dpunkt.verlag, 2012

[3] Martin, R. C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, mitp Professional, 2018