- GUI windows
- reports
- speech interface
- HTML, XML, XSLT, JSP, Javascript, ...

UI
(AKA Presentation, View)

- handles presentation layer requests
- workflow
- session state
- window/page transitions
- consolidation/transformation of disparate data for presentation

Application
(AKA Workflow, Process, Mediation, App Controller)

- handles application layer requests
- implementation of domain rules
- domain services (POS, Inventory)
  - services may be used by just one application, but there is also the possibility of multi-application services

Domain
(AKA Business, Application Logic, Model)

- very general low-level business services used in many business domains
- CurrencyConverter

Business Infrastructure
(AKA Low-level Business Services)

- (relatively) high-level technical services and frameworks
- Persistence, Security

Technical Services
(AKA Technical Infrastructure, High-level Technical Services)

- low-level technical services, utilities, and frameworks
- data structures, threads, math, file, DB, and network I/O

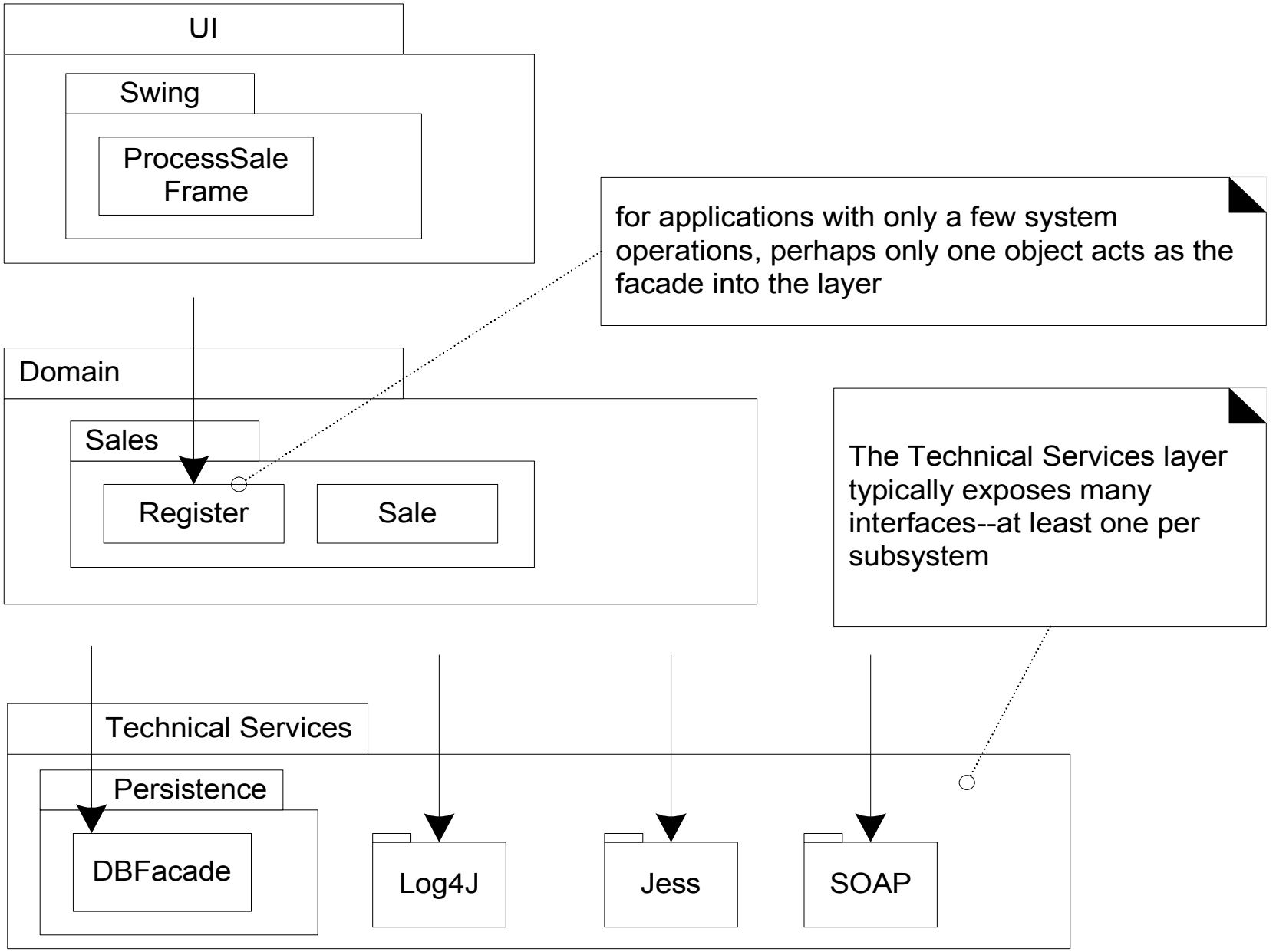
Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)

more app specific

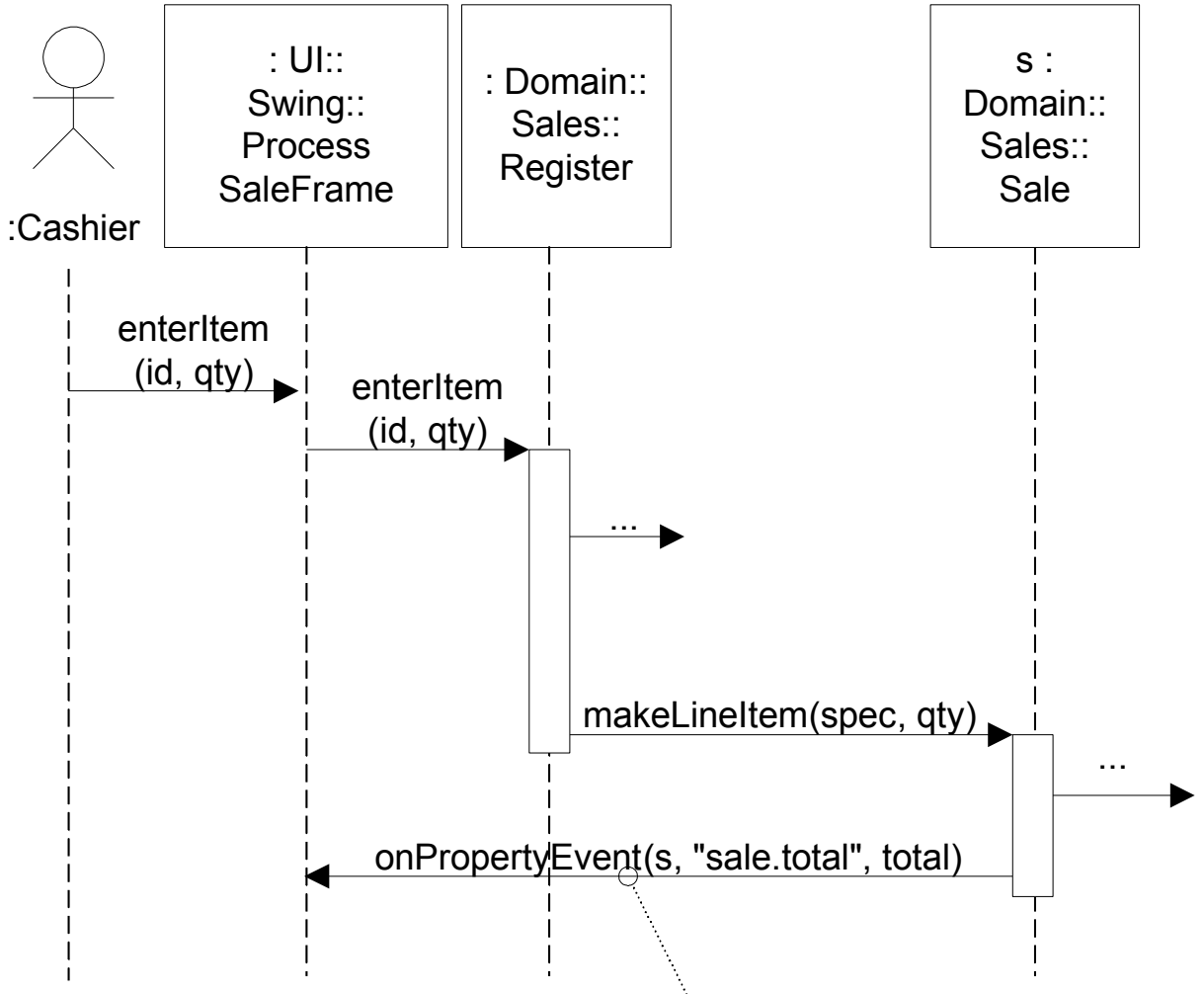
dependency

width implies range of applicability

Facade Controller

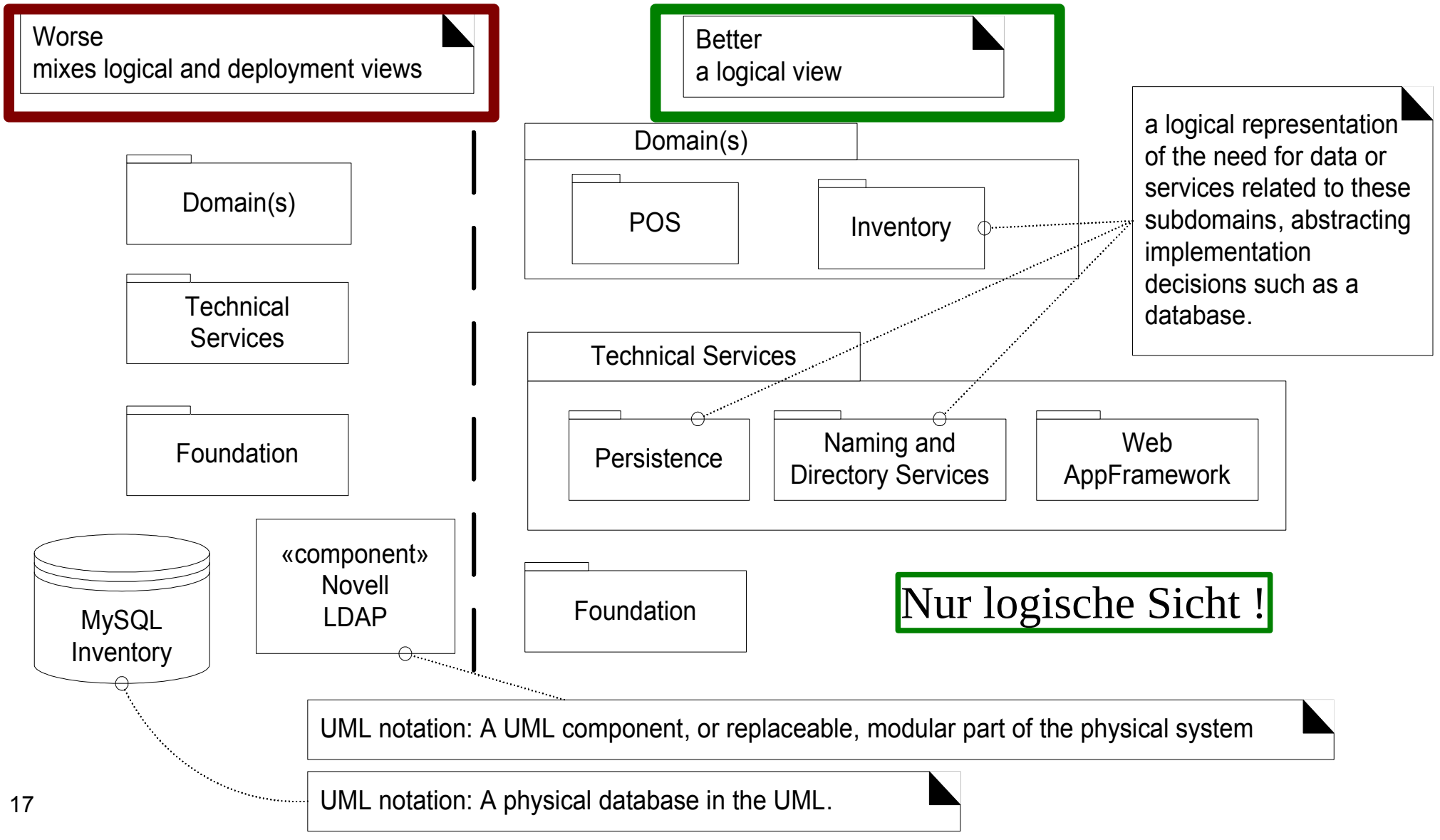


Observer Pattern



Collaboration from the lower layers to the UI layer is usually via the Observer (Publish-Subscribe) pattern. The Sale object has registered subscribers that are PropertyListeners. One happens to be a Swing GUI JFrame, but the Sale does not know this object as a GUI JFrame, but only as a PropertyListener.

Richtlinie für externe Ressourcen



Variations- und Entwicklungspunkte

- Variationspunkte („sicher“)
  - Varianten des aktuellen Systems oder der Anforderungen
- Entwicklungspunkte („spekulativ“)
  - Punkte, an denen eventuell später Änderungen auftreten können

Qualtitätsszenarios

Bevorzugt nichtfunktionale Anforderungen mit Szenario definieren :
<Wenn> ... <dann>
Kurz
Messbar, inklusive Spezifikation der Messumgebung
Auf erfolgsrelevante Faktoren beschränken

Nicht-Funktionale Anforderungen

Nicolas 13.10.2016, 07:48:26

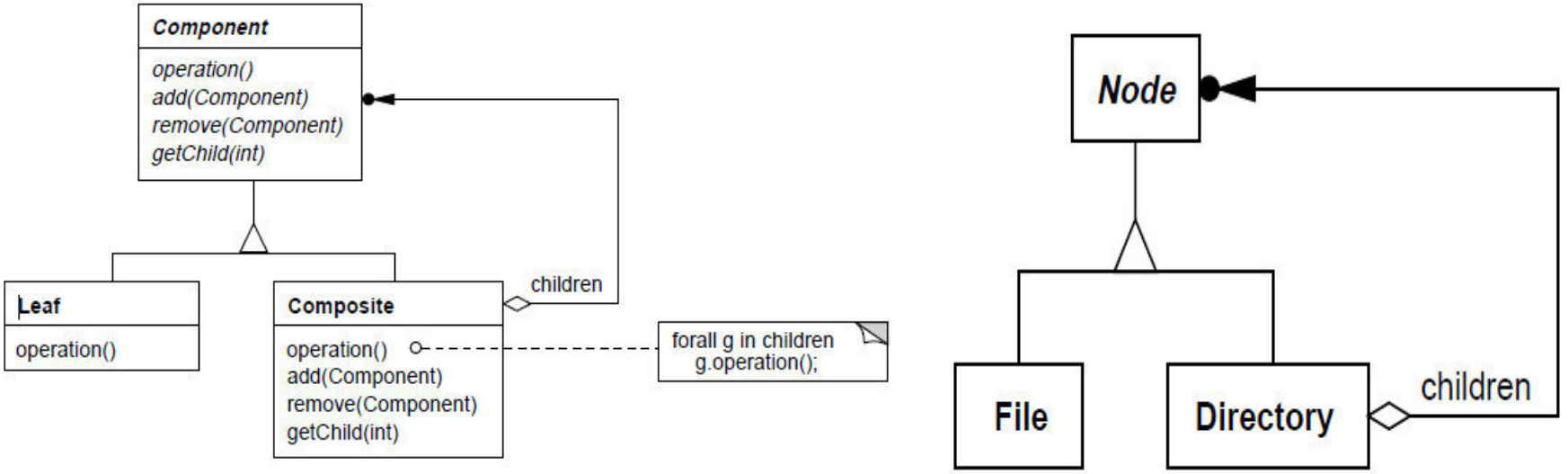
F - bei Feuer kein Betrieb
U - behindertengerecht (Blindenschrift...)
- Priorität
R - 24/7
P - Reaktion < 30sec
S - Einfacher Zugang für Wartungstechniker
+ - Musik, Spiegel, Standards

Wir fragen uns, welche nicht-funktionalen Anforderungen eine Liftsteuerung erfüllen muss. Wer bedient und benutzt einen Lift? Was wird transportiert? Was passiert, wenn etwas schief geht?

Aufgabe :
Schreiben Sie die wichtigsten nicht-funktionalen Anforderungen einer Liftsteuerung auf.
Zeit : 5 Min

Anschliessend gehen wir durch die verschiedenen Kategorien der nicht-funktionalen Anforderungen durch und besprechen diese.

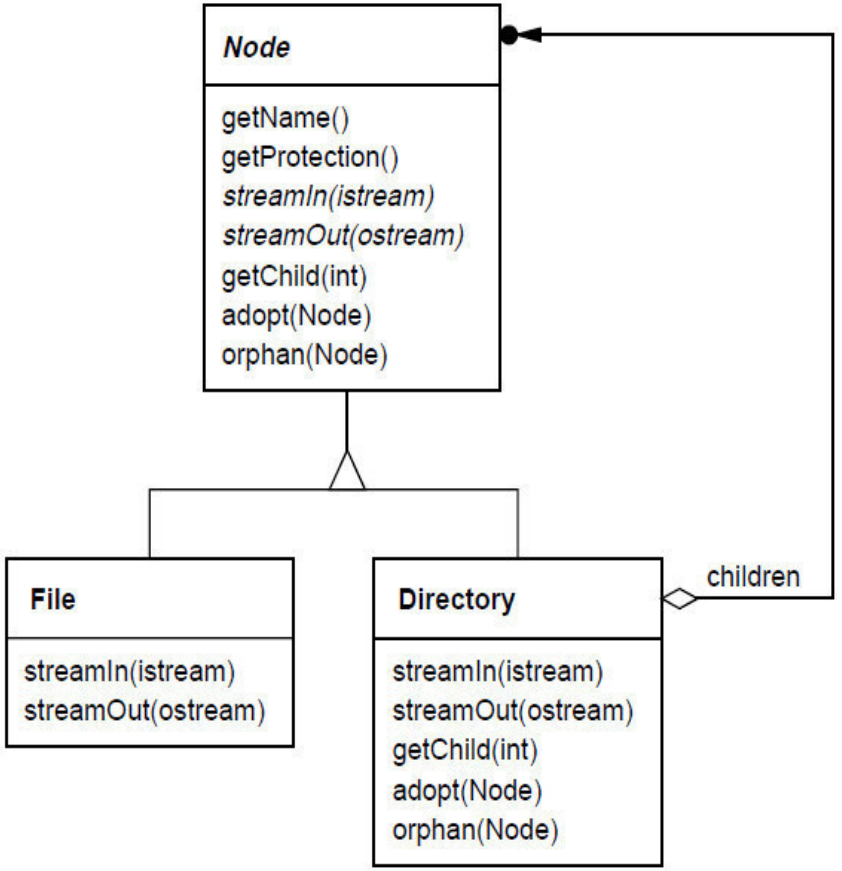
Composite Pattern (1)



Lösung:
Rekursive Baumstruktur → *Composite Pattern*

Was sind die Konsequenzen bzw. Resultate und der Trade-off bei Verwendung des Pattern?

Composite Pattern (2)



What *uniform interface* does `Node` define?

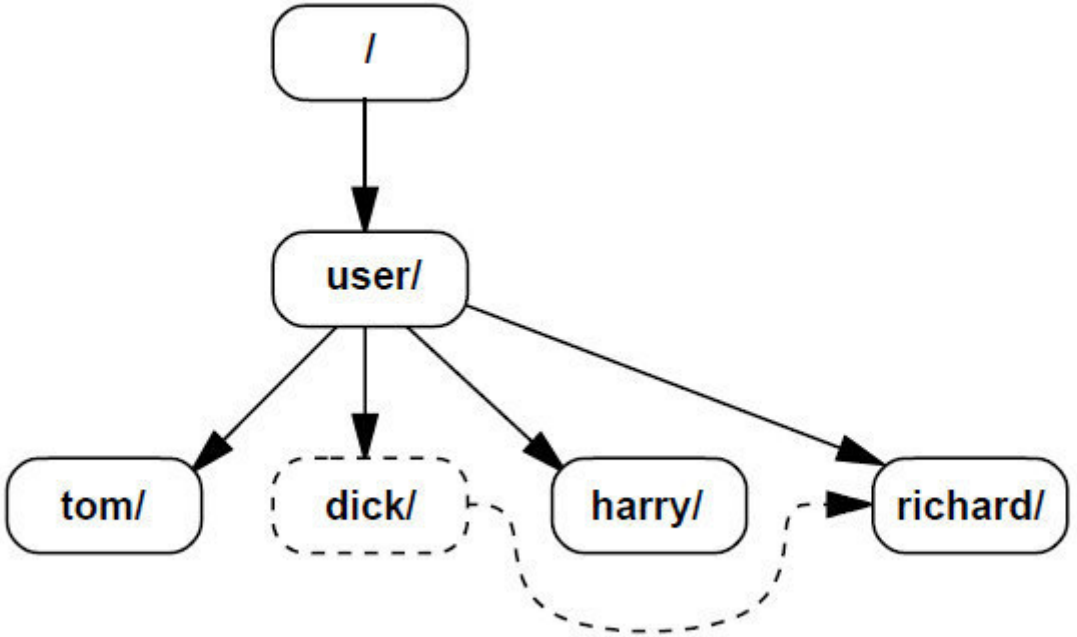
- get name/protection
  - Obviously common
- stream in/out
  - Less-obviously common
- enumerate children
  - Needed for recursion, hiding internal data structure
  - Could apply ITERATOR instead
- adopt & orphan
  - Trade-off between type safety and uniformity

(Ownership eines Nodes übernehmen bzw. abgeben)

Diskussion (3)

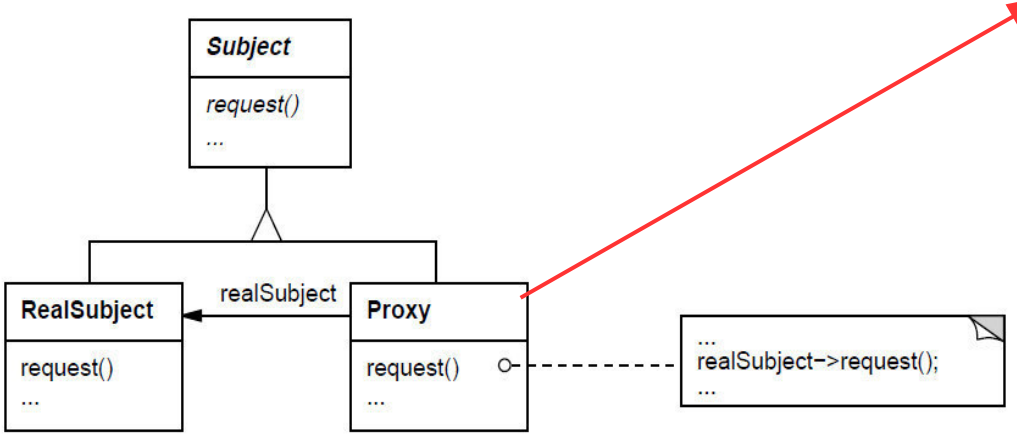
- „mkdir“ ist Funktion und keine Methode von Directory (oder Node)
  - Wieso?
    - Kompakte minimale API, die stabil ist („Unix-Style“)
  - Aber soll jeder Verwender des File-Systems diese Funktionalität wieder von Grund auf selber programmieren?
  - Wäre denn die Klasse weniger kohäsiv, wenn sie mkdir (oder createSubDirectory(…)) selber anbieten würde?

Weitere Anforderungen: Symbolische Links



Problem:
Wie füge ich nicht invasiv symbolische Links zum Dateisystem hinzu, die über Directories, Dateisysteme und sogar verschiedene Computer funktionieren?

Proxy Pattern (1)




```
public class Link extends Node {
    public Node getChild (int n) {
        return _subject.getChild(n);
    }
    // ...
```

Additional Link-specific operation:

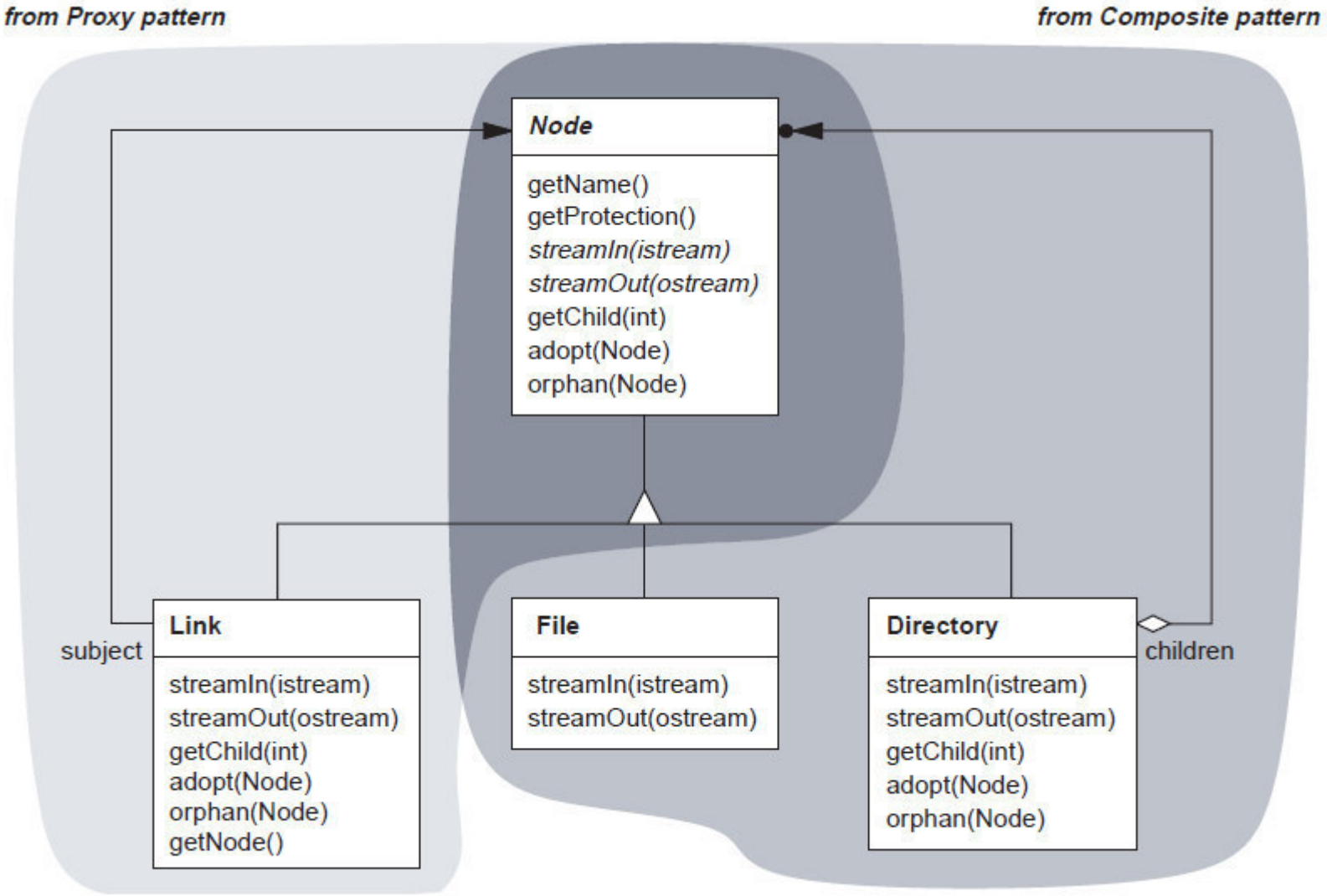
```
Node getNode () { return _subject; }
```

(for clients who know they are dealing with a `Link`)

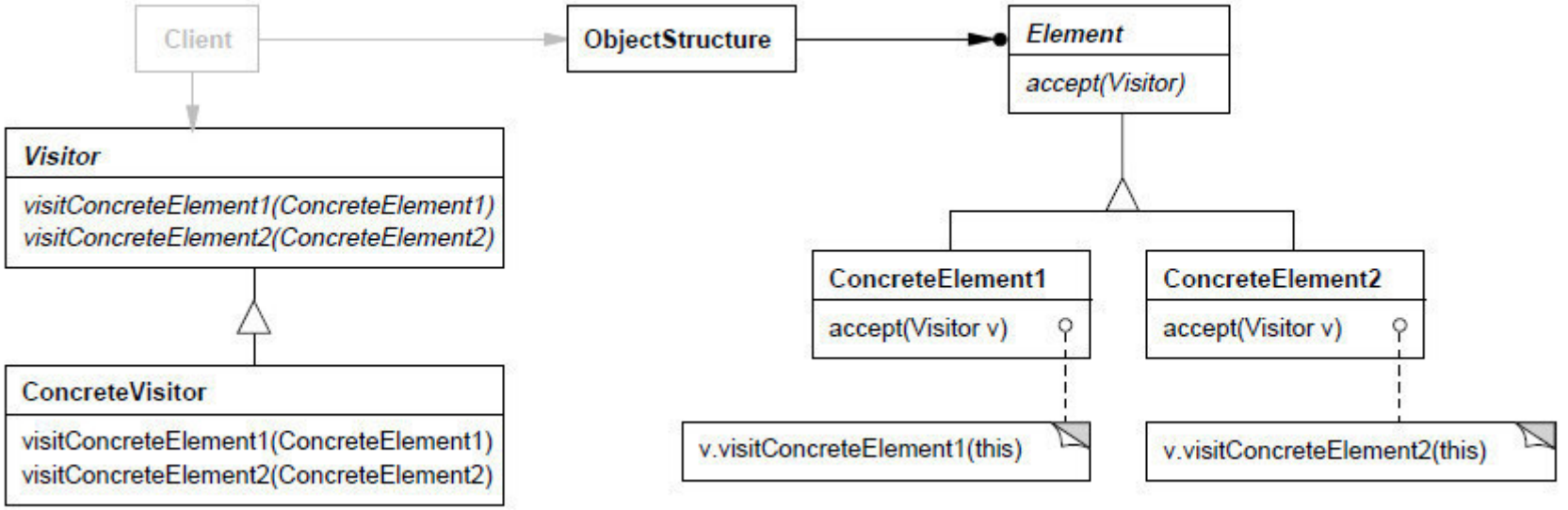
Lösung:
Lässt Variation zu, wie auf ein Objekt zugegriffen wird und wo es lokalisiert ist → *Proxy*.

Was sind die Konsequenzen bzw. Resultate und der Trade-off bei Verwendung des Pattern?

Dateisystem: Zwischenstand



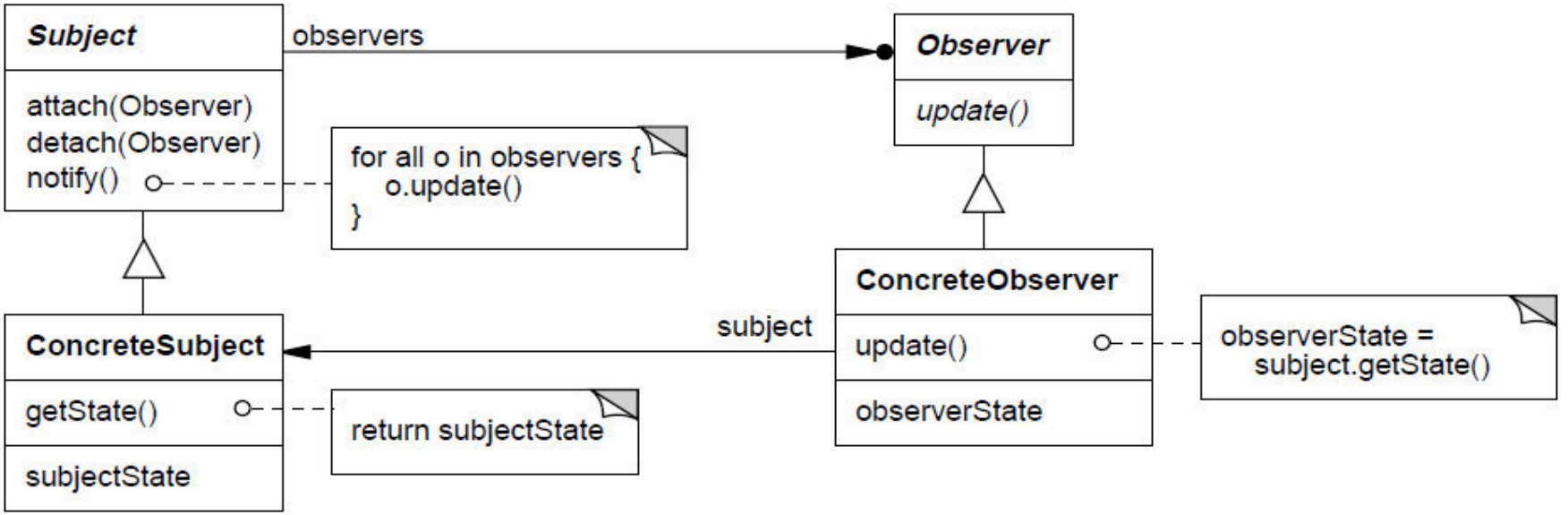
Visitor Pattern (1)



Lösung:
Externalisierung von Operationen auf eine Objekt-Struktur → *Visitor*.

Was sind die Konsequenzen bzw. Resultate und der Trade-off bei Verwendung des Pattern?

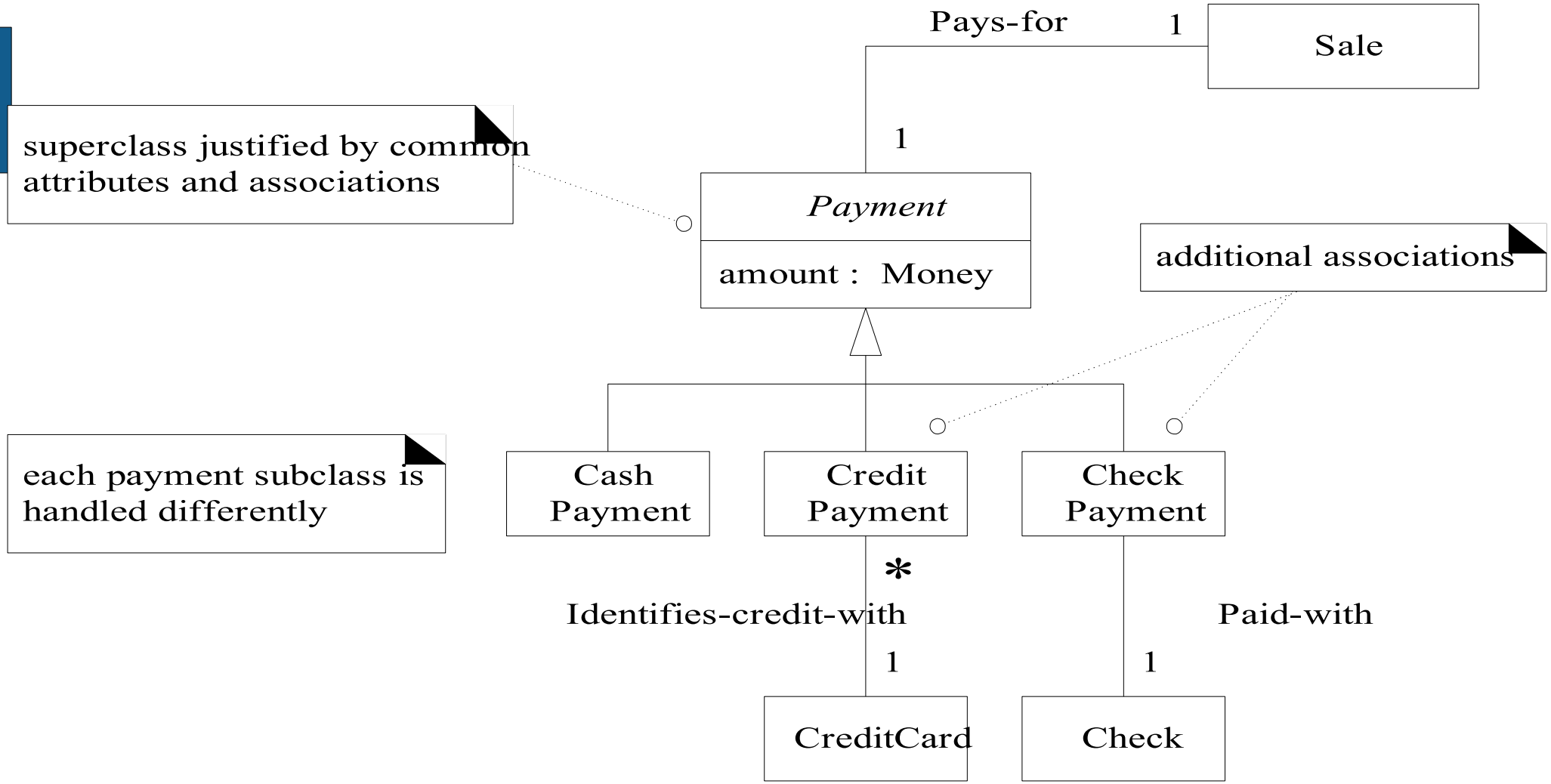
Observer Pattern (1)



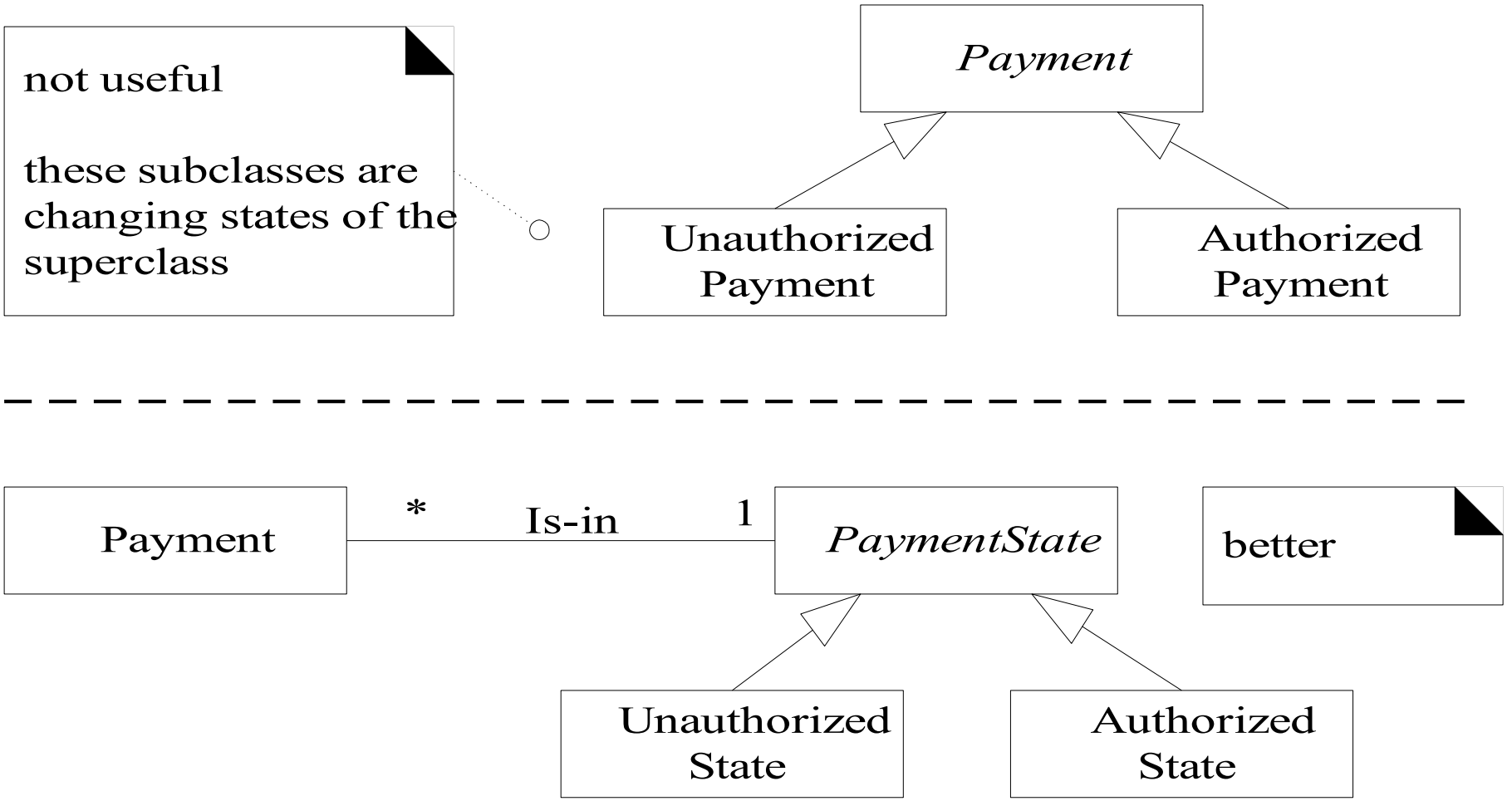
Lösung:
Wenn ein Objekt ändert, werden alle abhängigen Objekte benachrichtigt (Update) → *Observer*.

Was sind die Konsequenzen bzw. Resultate und der Trade-off bei Verwendung des Pattern?

Fallstudie

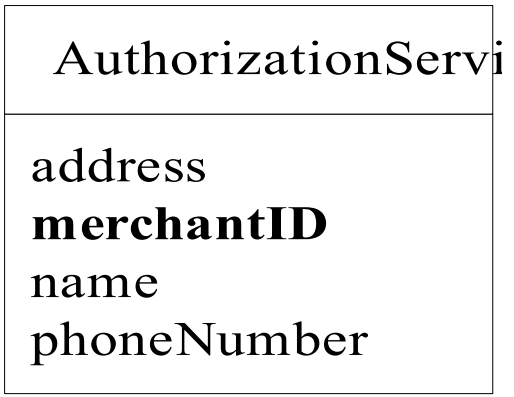

Pays-for
1
Sale
1
superclass justified by common attributes and associations
Payment
amount : Money
additional associations
each payment subclass is handled differently
Cash Payment
Credit Payment
Check Payment
*
Identifies-credit-with
Paid-with
1
1
CreditCard
Check

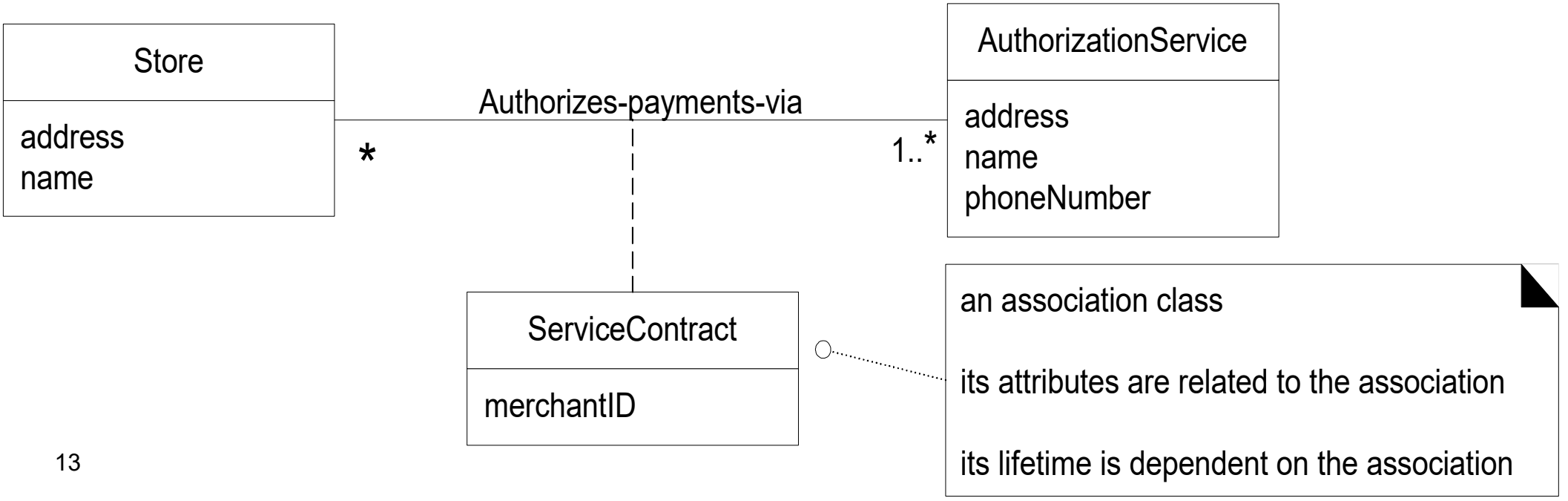
Zustände modellieren ?



Assoziationsklassen

both placements of merchantID are incorrect because there may be more than one merchantID

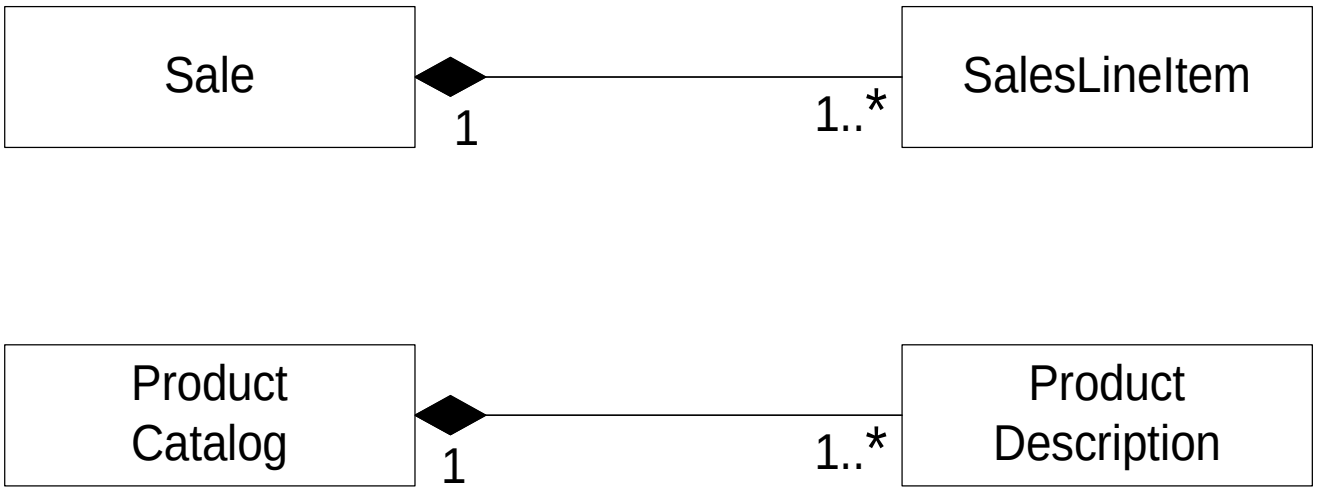




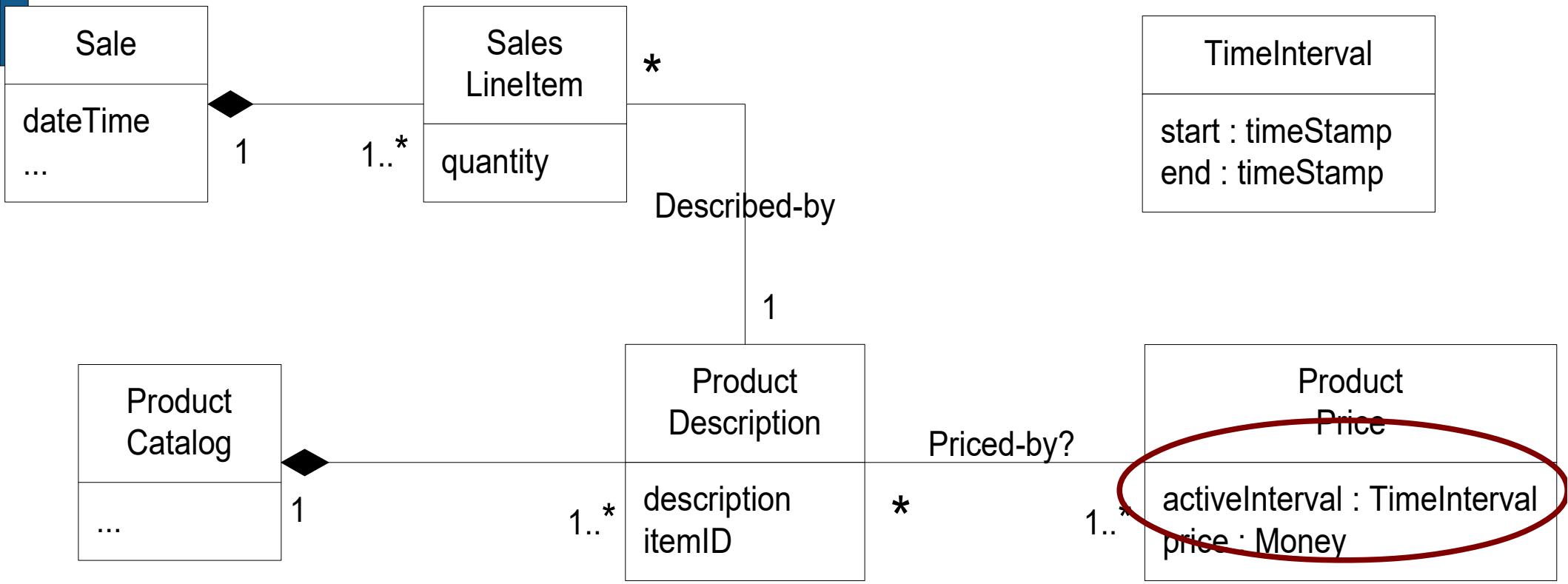
Aggregation und Komposition

- Aggregation (leere Raute) ist „Modellierungsplazebo“
- Komposition (ausgefüllte Raute) ist starke „Teil-Ganzes“ Beziehung
  - Lebensdauer des Teil ist begrenzt vom Ganzen
  - Physischer oder logischer Teil
  - Gewisse Eigenschaften werden weitergegeben
  - Operationen werden weitergegeben
- Empfehlung : nur Kompositionen modellieren

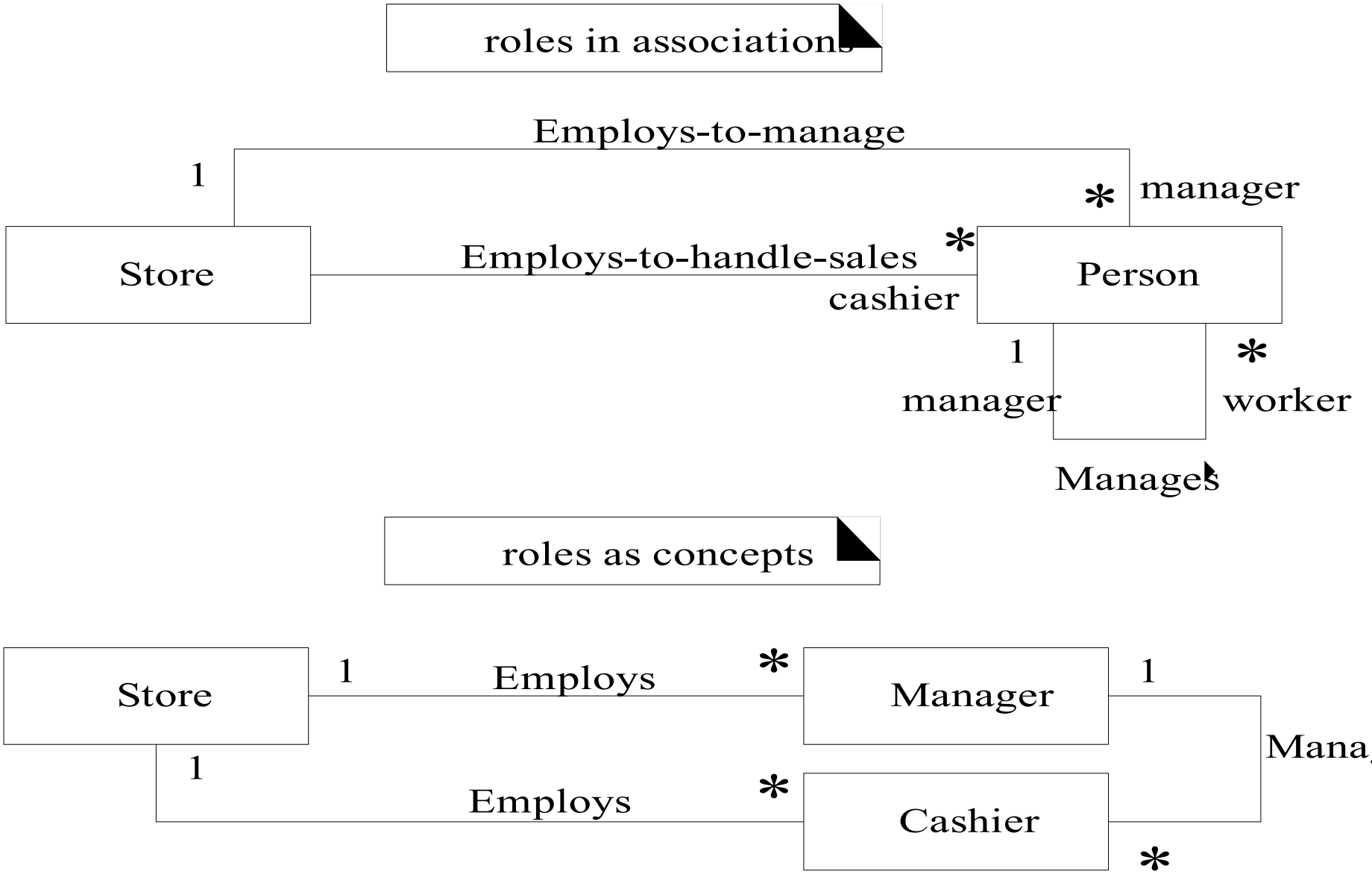
Komposition Fallstudie



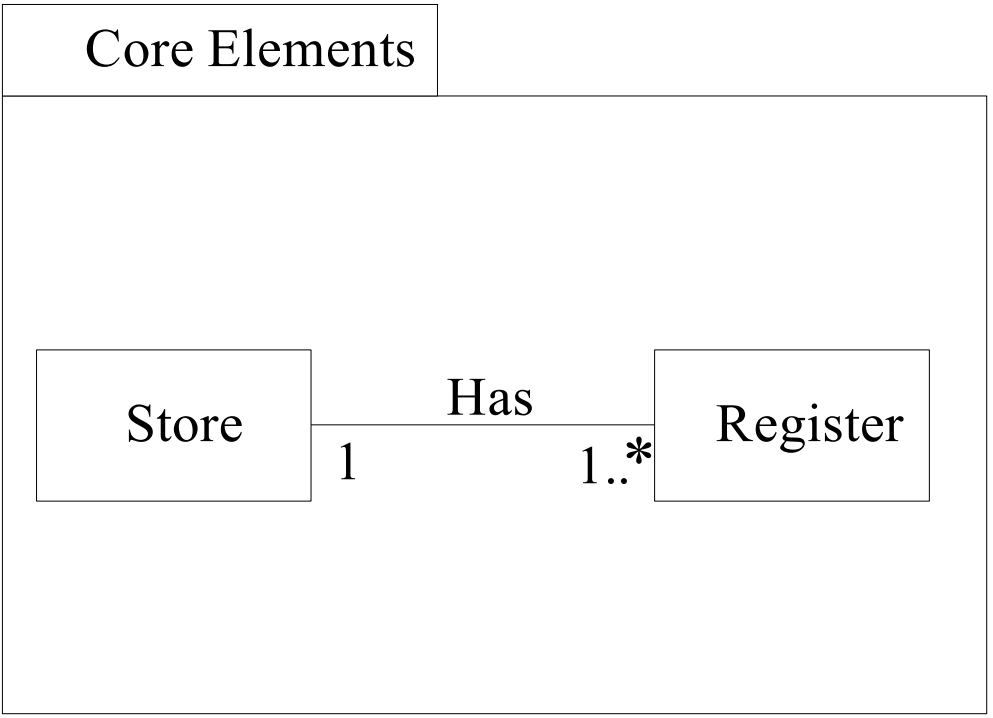
Fallstudie, Zeitintervalle

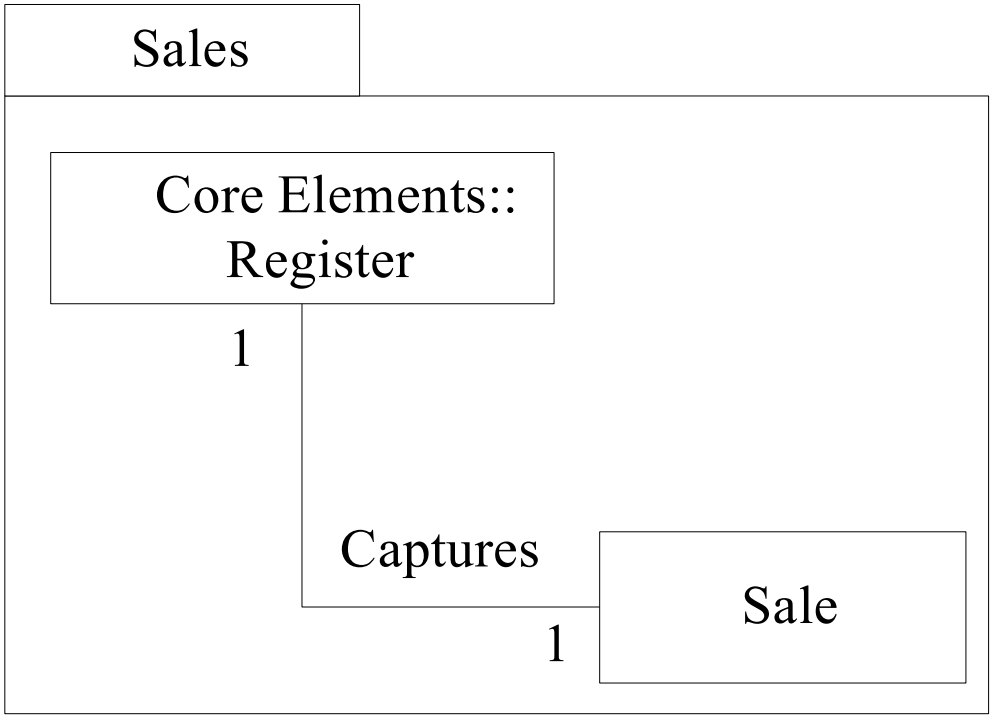


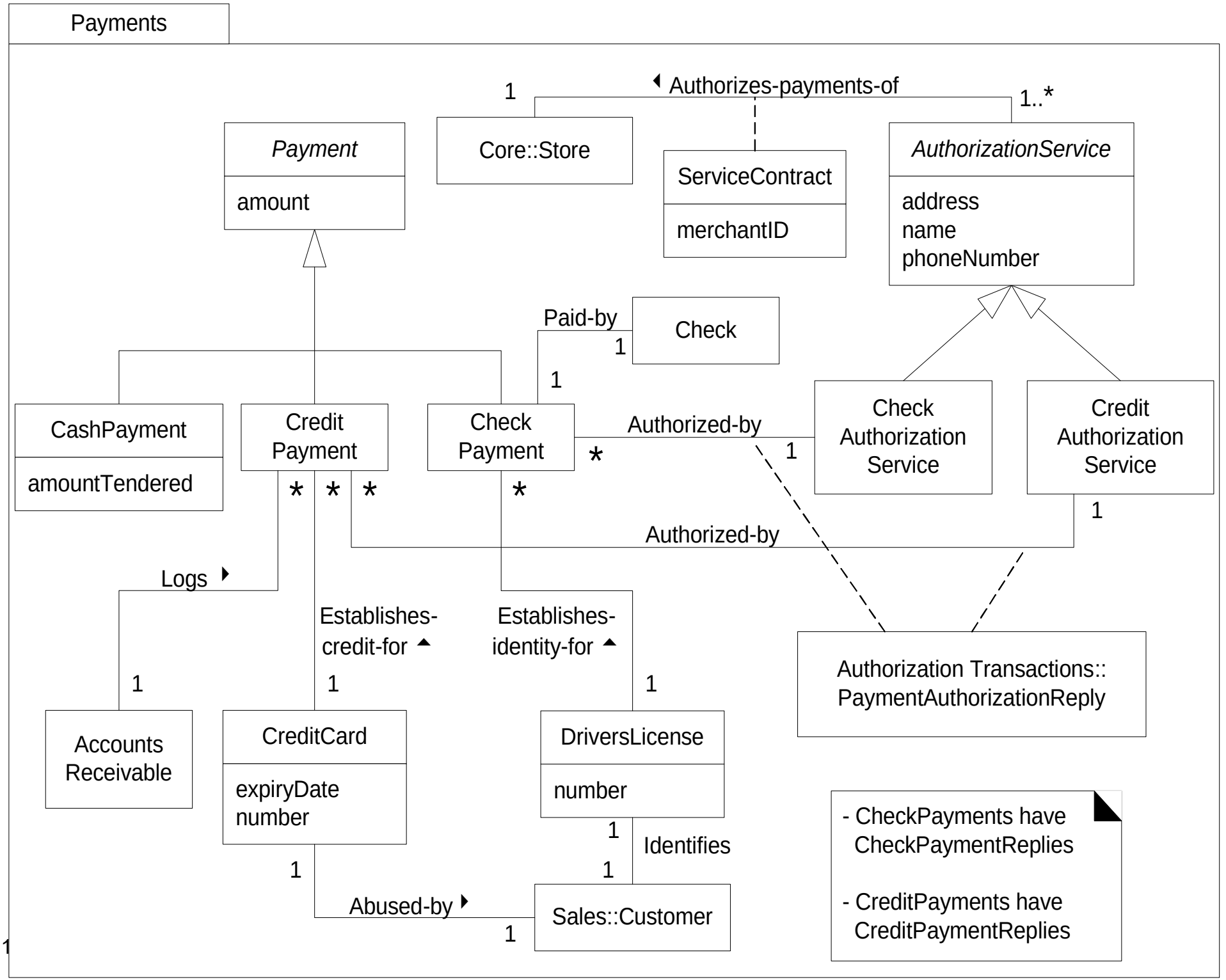
Rollen modellieren

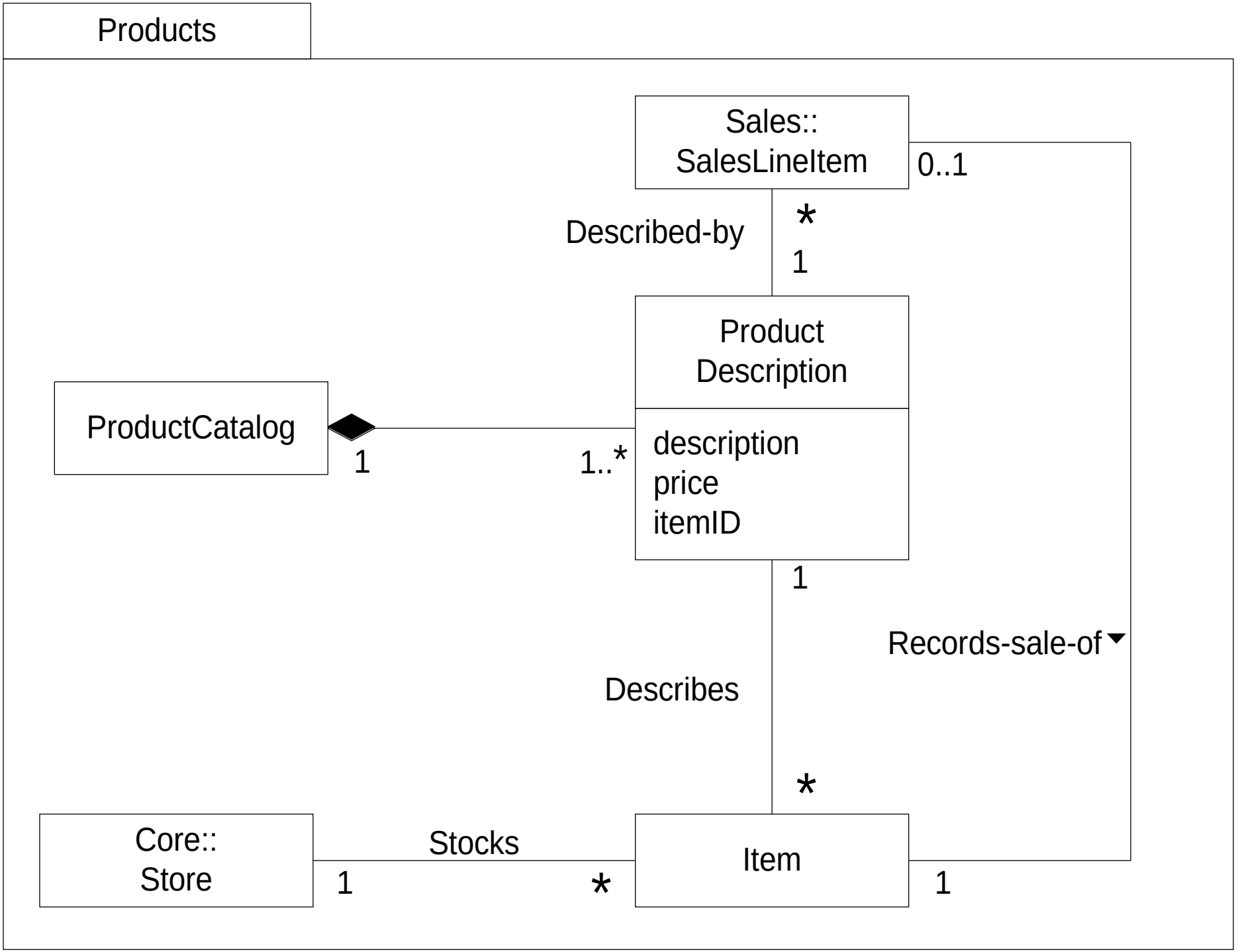


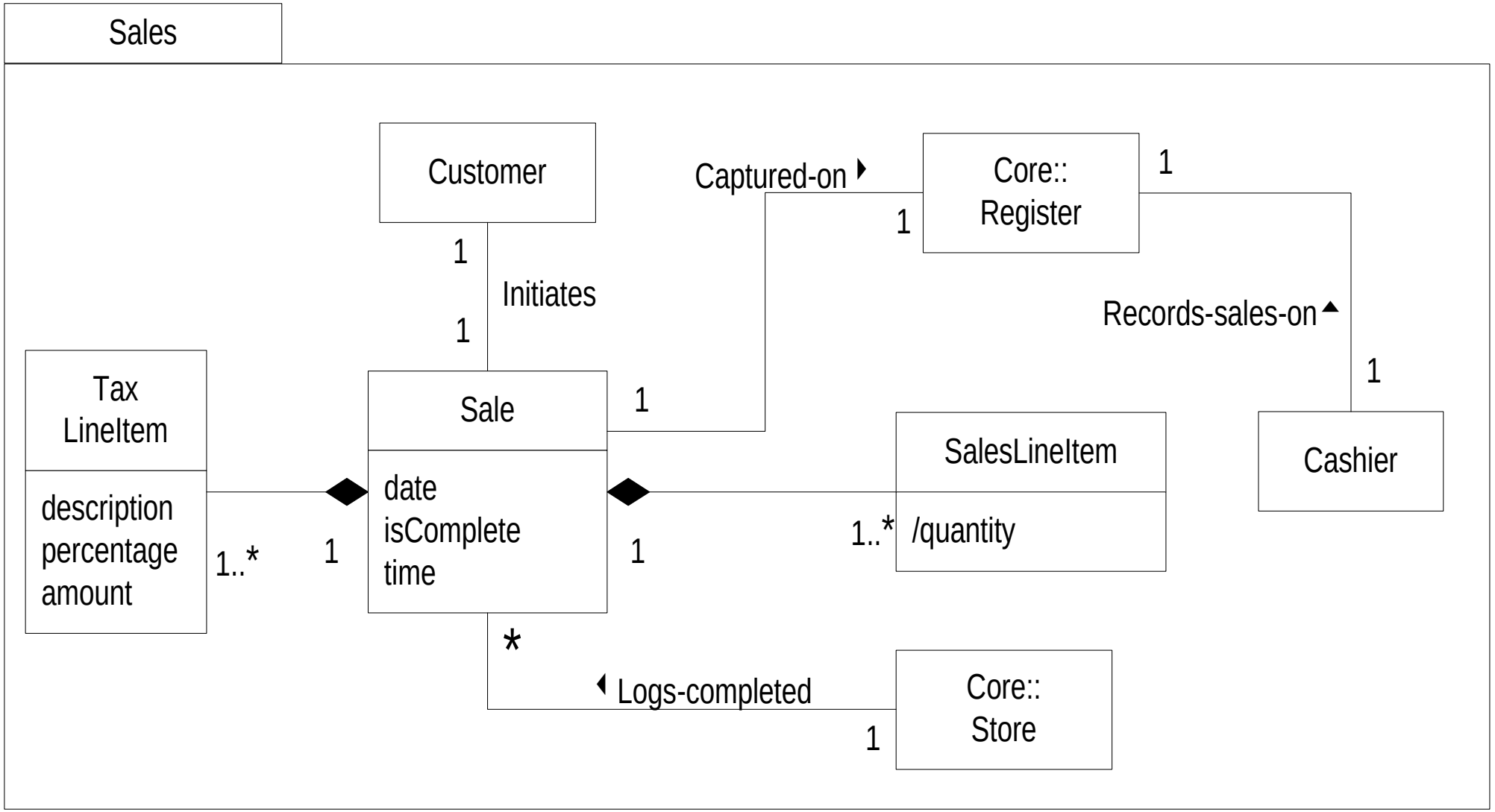
Referenzen auf andere Pakete






Payments
Authorizes-payments-of
1
1..*
Payment
amount
Core::Store
ServiceContract
merchantID
AuthorizationService
address
name
phoneNumber
Paid-by
Check
1
1
CashPayment
amountTendered
Credit Payment
Check Payment
Authorized-by
*
1
Check Authorization Service
Credit Authorization Service
1
*
*
*
*
Authorized-by
Logs
Establishes-credit-for
Establishes-identity-for
1
1
1
Accounts Receivable
CreditCard
expiryDate
number
DriversLicense
number
Authorization Transactions:: PaymentAuthorizationReply
1
1
Identifies
1
1
Abused-by
Sales::Customer
1
- CheckPayments have CheckPaymentReplies
- CreditPayments have CreditPaymentReplies


Products
Sales::
SalesLineItem
0..1
Described-by
*
1
Product
Description
ProductCatalog
1
1..*
description
price
itemID
1
Records-sale-of
Describes
*
Core::
Store
Stocks
1
*
Item
1

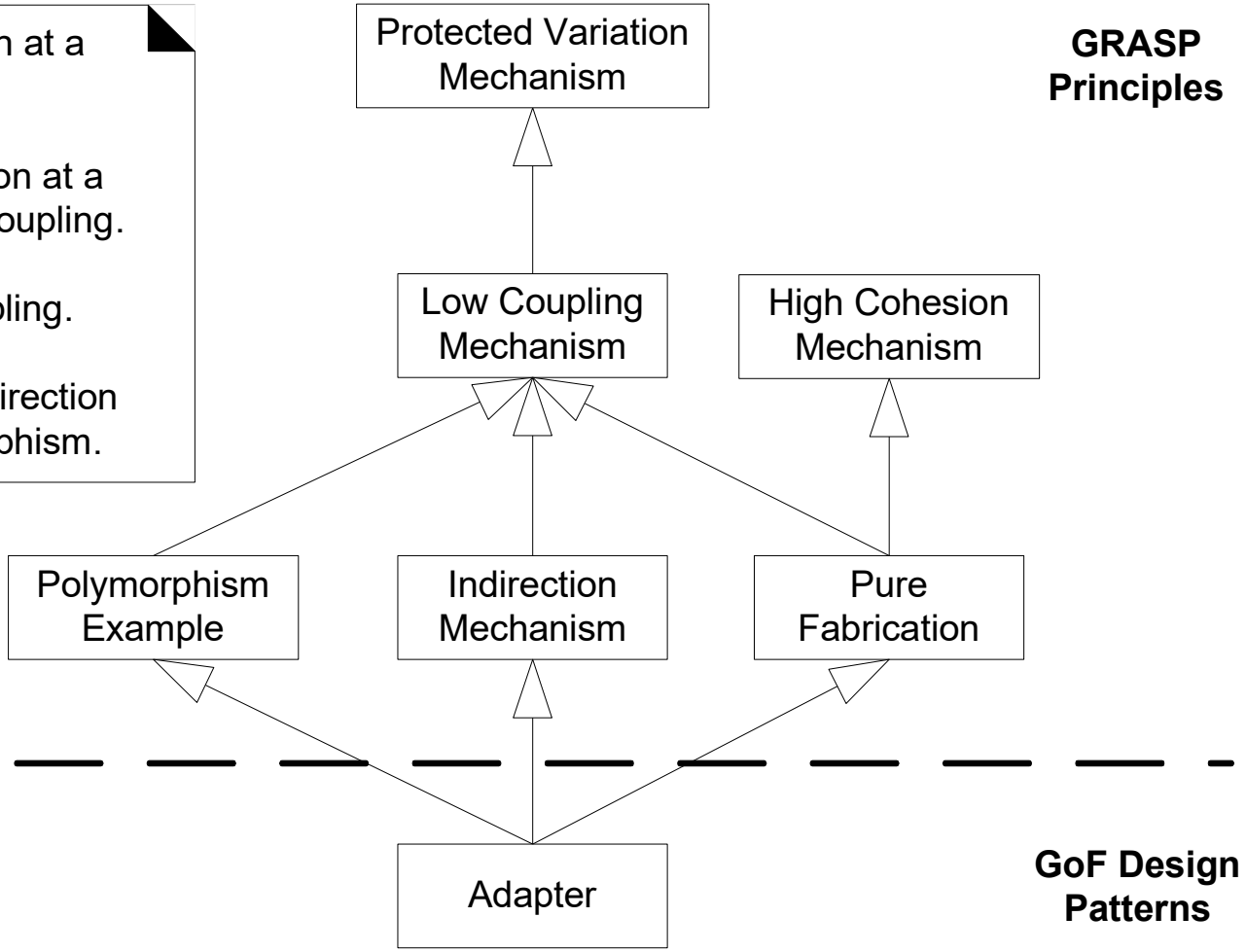

Sales
Customer
Captured-on
Core::
Register
1
1
1
Initiates
1
1
Records-sales-on
1
Tax
LineItem
description
percentage
amount
Sale
date
isComplete
time
1
SalesLineItem
/quantity
Cashier
1..*
1
1
1..*
*
Logs-completed
Core::
Store
1

Low coupling is a way to achieve protection at a variation point.

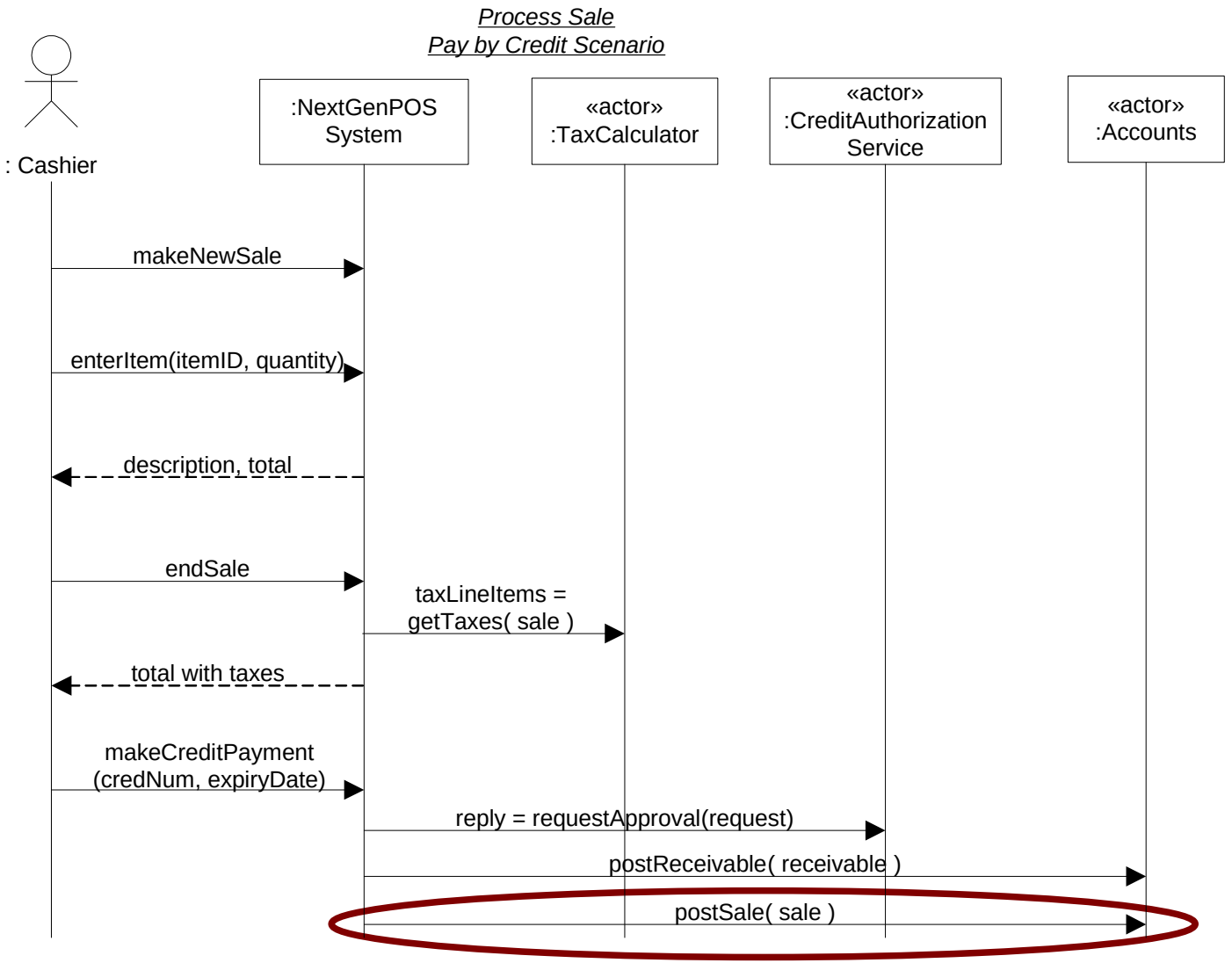
Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.

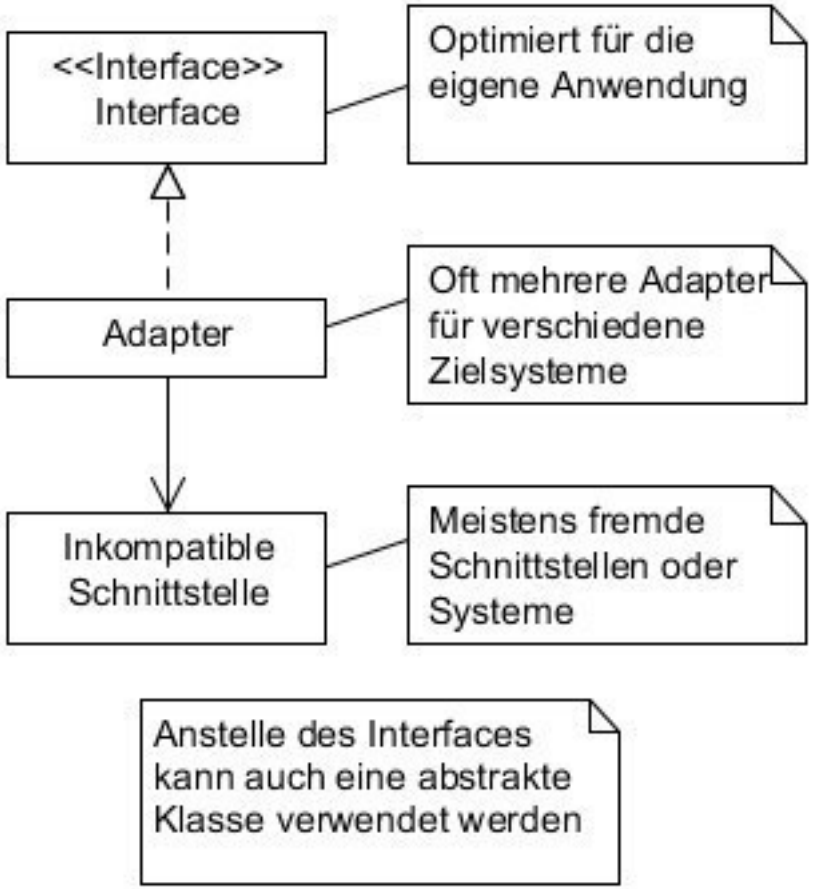


Repetition SSD



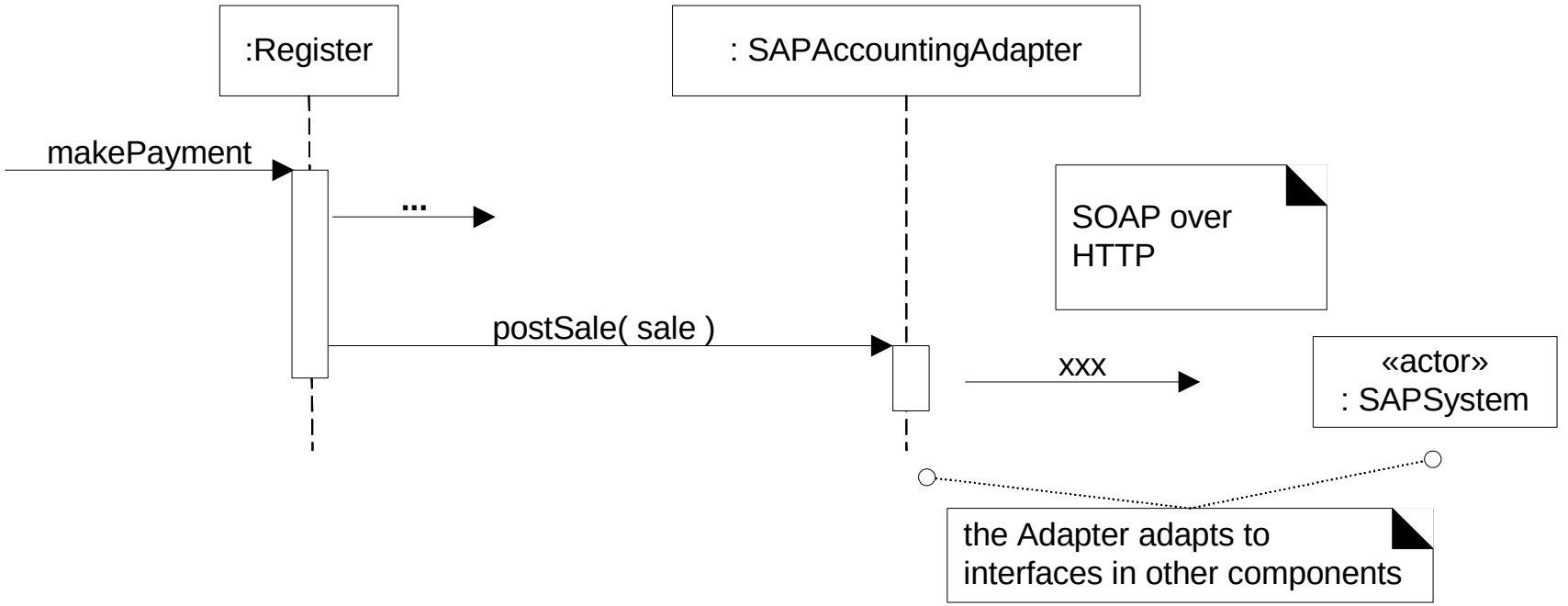
Adapter Pattern

- Problem
  - Wie können unterschiedliche Schnittstellen angepasst werden ?
- Lösung
  - Adapterklasse dazwischenschalten, die die Anpassung vornimmt



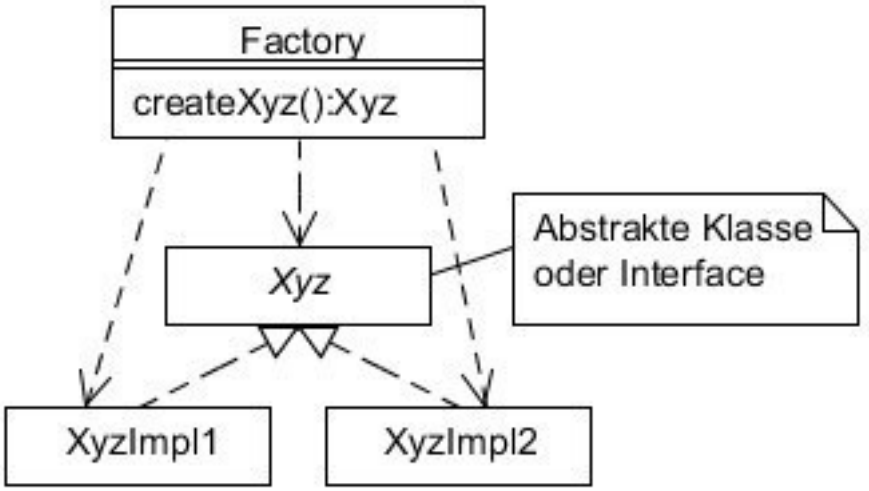
Lösung (1)

- Adapter einsetzen für Anpassung an externen Dienst



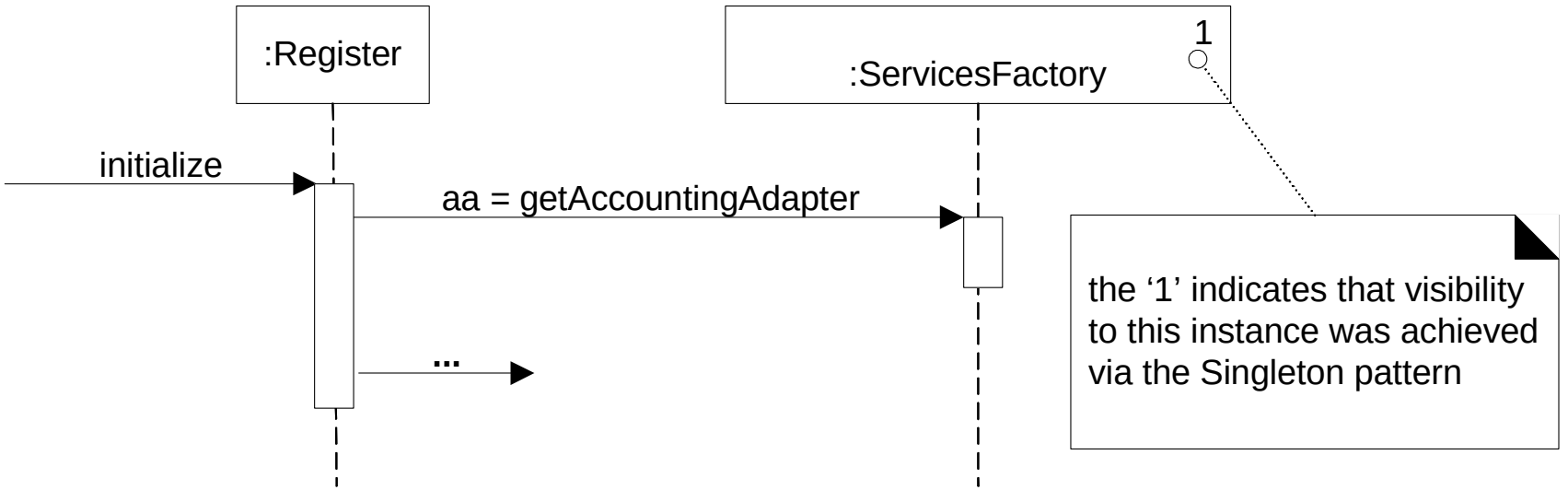
Factory Pattern

- Problem
  - Erstellung eines Objekts ist komplex
- Lösung
  - Eine spezielle Klasse, die die einzige Aufgabe hat, Objekte einer anderen Klasse zu erzeugen



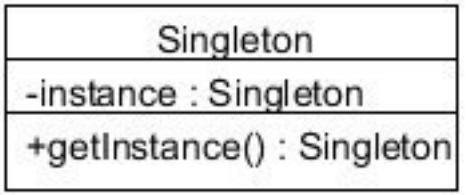
Lösung (2)

- Wer erzeugt Adapter ?
  - Factory Pattern einsetzen
- Zugriff auf Factory ?
  - Singleton

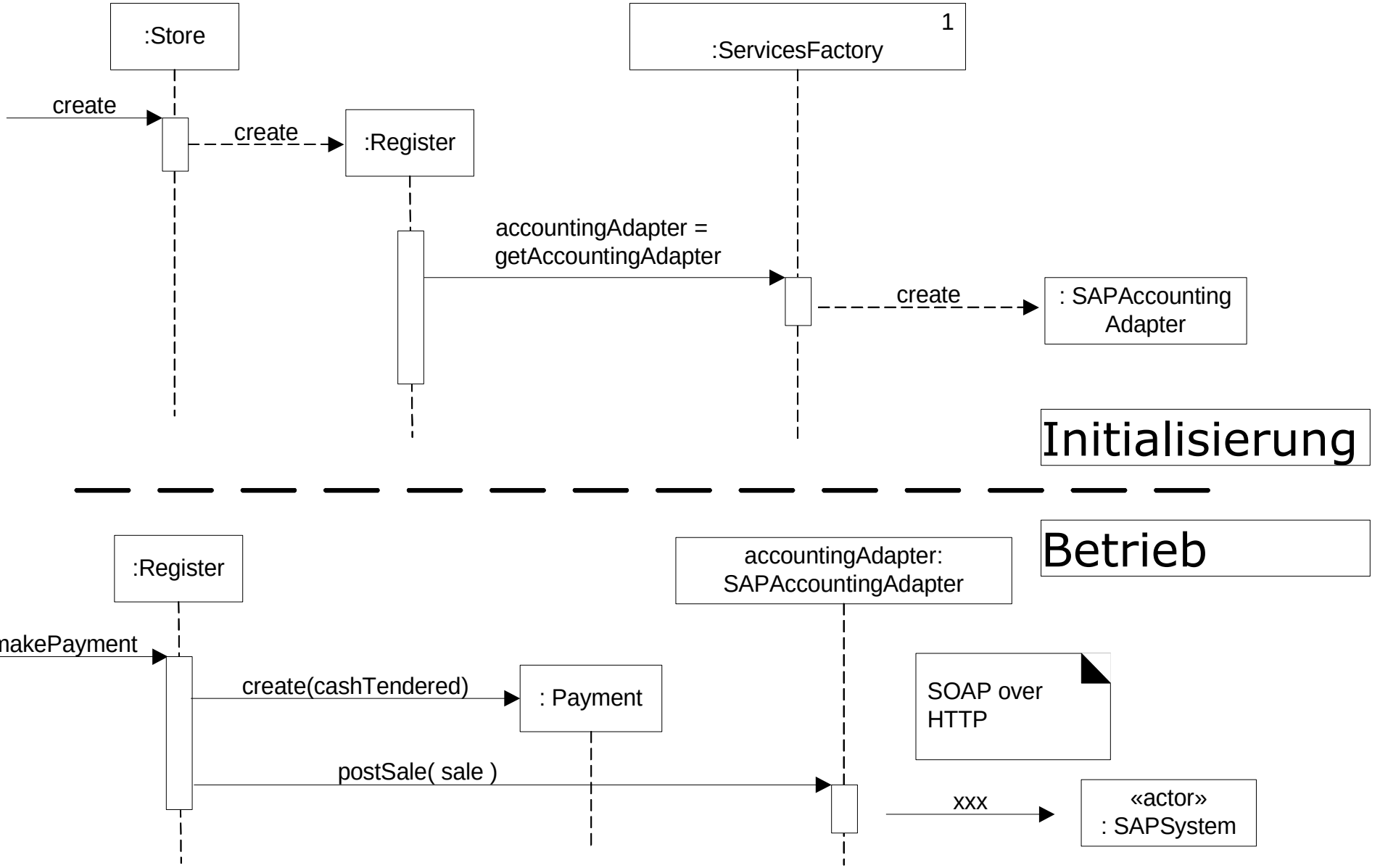


Singleton Pattern

- Problem
  - Sie benötigen von einer Klasse nur genau 1 Instanz
  - Diese Instanz sollte global sichtbar sein
- Lösung
  - Statische Methode, die immer dasselbe Objekt zurückgibt
  - Eventuell wird dieses Objekt erst beim ersten Zugriff erzeugt

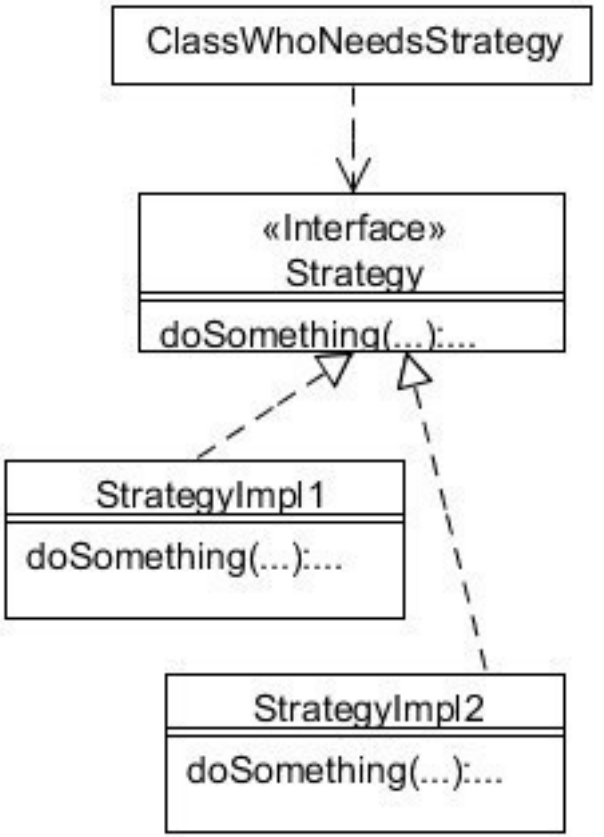


Gesamtlösung



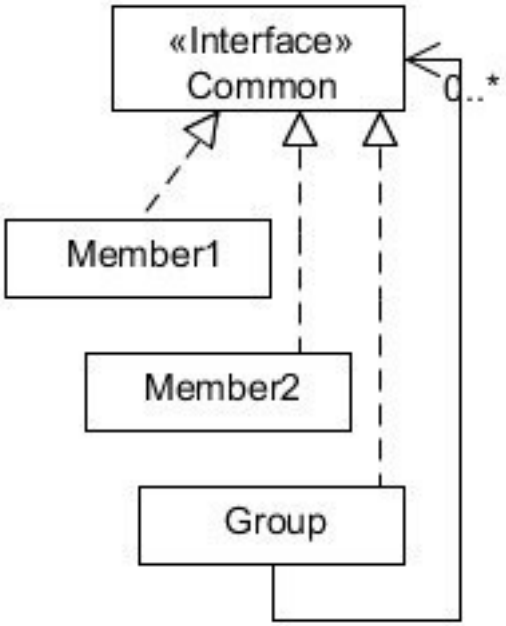
Strategy Pattern

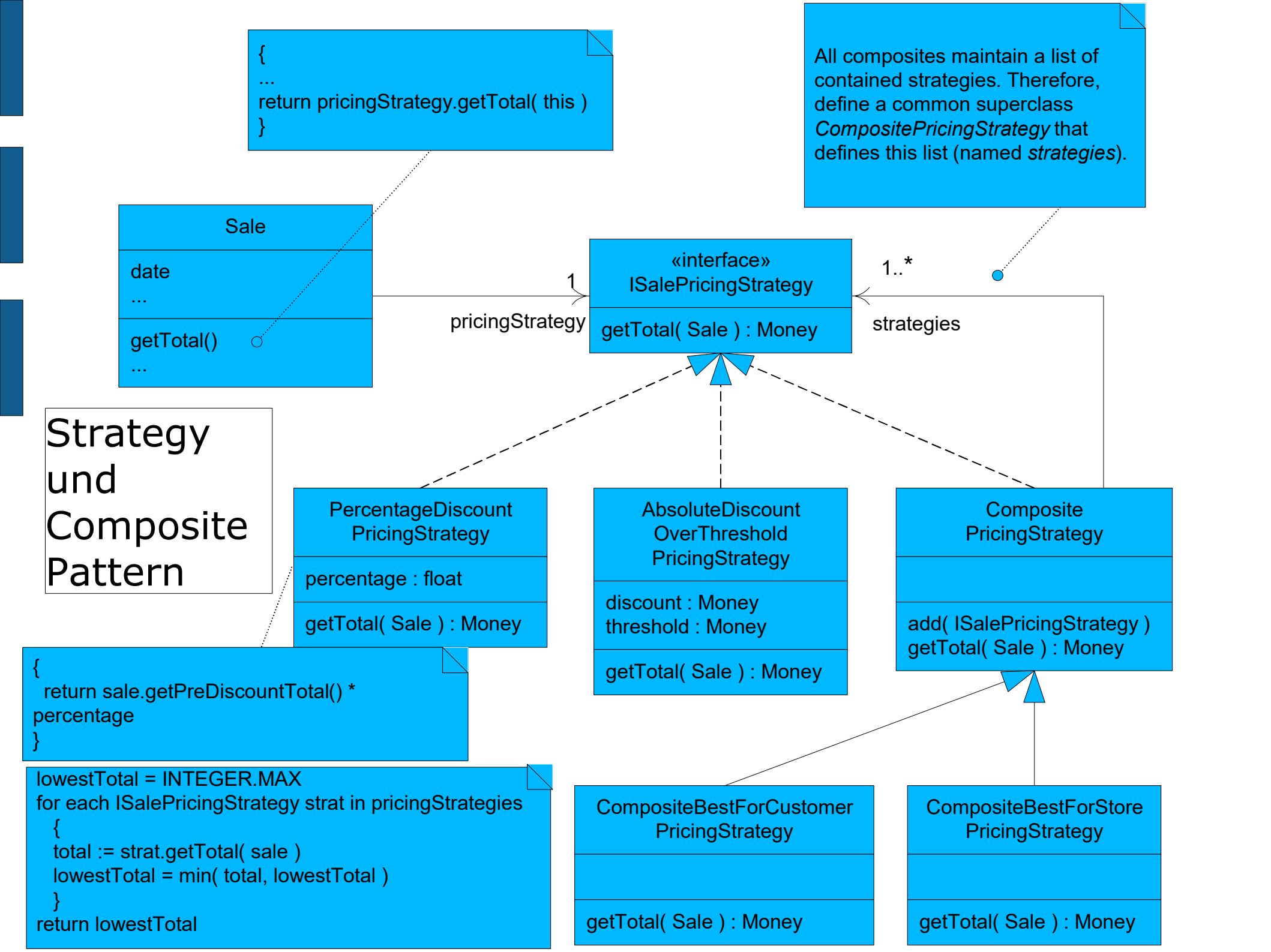
- Problem
  - Einfache und schnelle Anpassung an unterschiedliche, aber verwandte Algorithmen oder Geschäftsregeln
- Lösung
  - Interface für die Anwendung des Algorithmus definieren und unterschiedliche Implementationen anbieten
  - Algorithmus wird in eine eigene Klasse ausgelagert



Composite Pattern

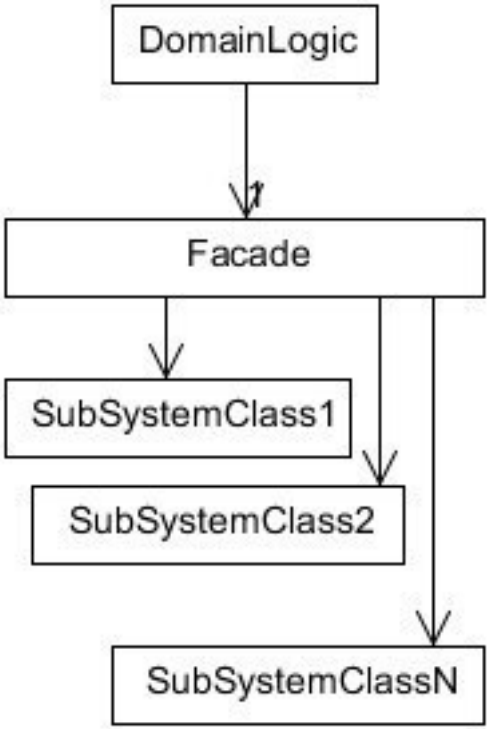
- Problem
  - Gruppe von Objekten soll gleich behandelt werden wie ein einzelnes Objekt
- Lösung
  - Eigene Klasse für die Gruppe von Objekten definieren, die dasselbe Interface implementiert wie die einzelnen Mitglieder




{
...
return pricingStrategy.getTotal(this)
}
All composites maintain a list of contained strategies. Therefore, define a common superclass *CompositePricingStrategy* that defines this list (named *strategies*).
Sale
date
...
getTotal()
...
1
pricingStrategy
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
1..*
strategies
Strategy und Composite Pattern
PercentageDiscount PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money
Composite PricingStrategy
add(ISalePricingStrategy)
getTotal(Sale) : Money
{
return sale.getPreDiscountTotal() *
percentage
}
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
{
total := strat.getTotal(sale)
lowestTotal = min(total, lowestTotal)
}
return lowestTotal
CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money
CompositeBestForStore PricingStrategy
getTotal(Sale) : Money

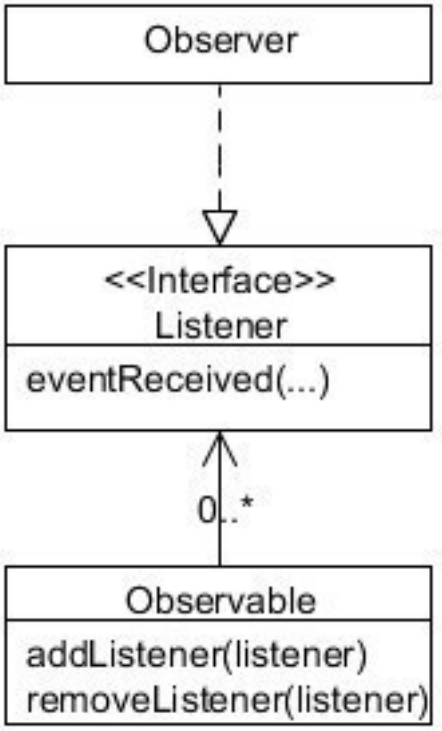
Facade Pattern

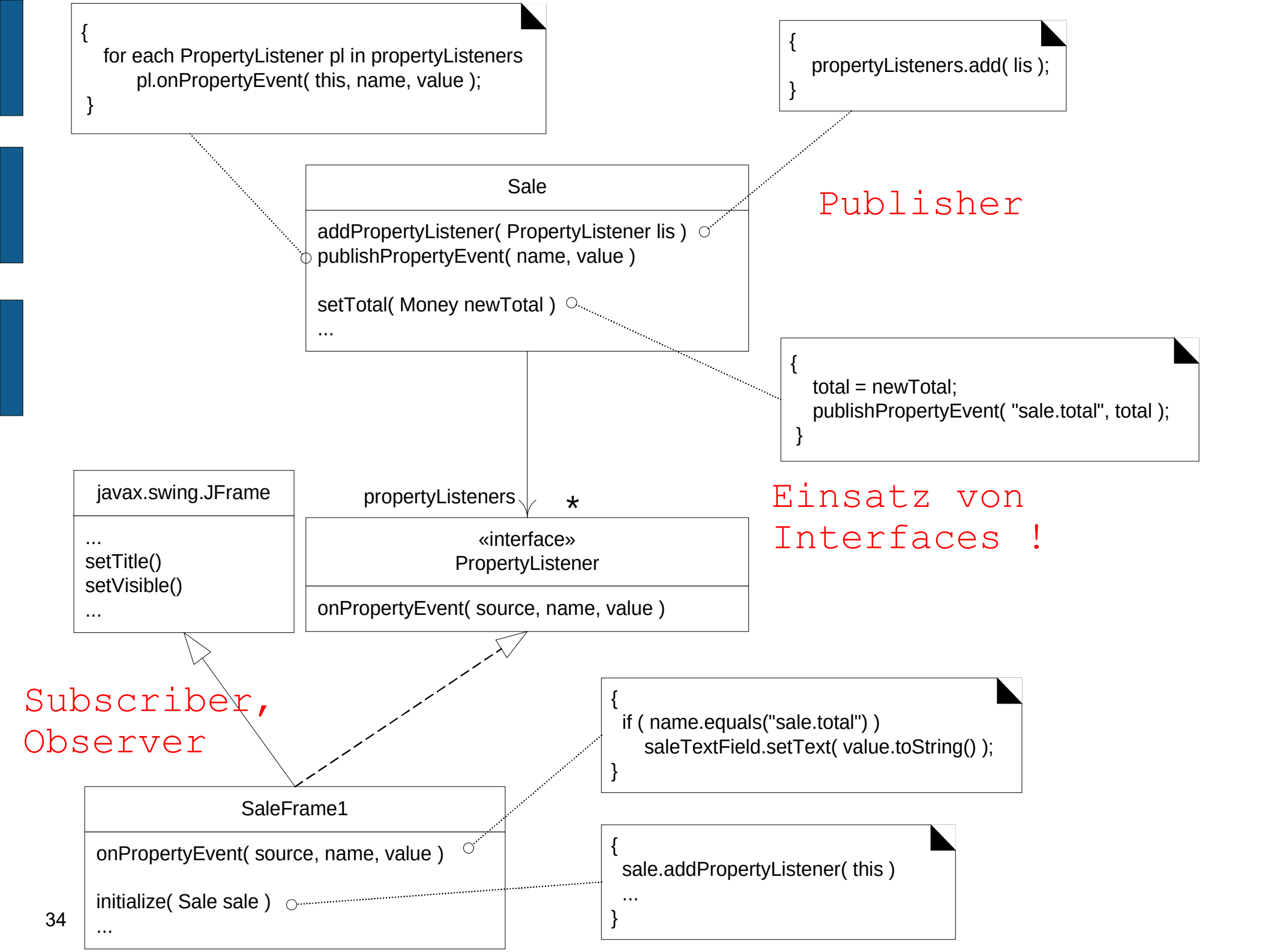
- Problem
  - Es wird ein einheitliches Interface zu einem ganzen Satz von Klassen (Subsystem) benötigt, so dass das Subsystem einfach ausgewechselt werden könnte
- Lösung
  - Eine einzige Klasse definieren, die den einzigen Kontaktpunkt mit dem Subsystem darstellt
  - Diese Klasse ist dann sozusagen die Fassade für das Subsystem



Observer Pattern

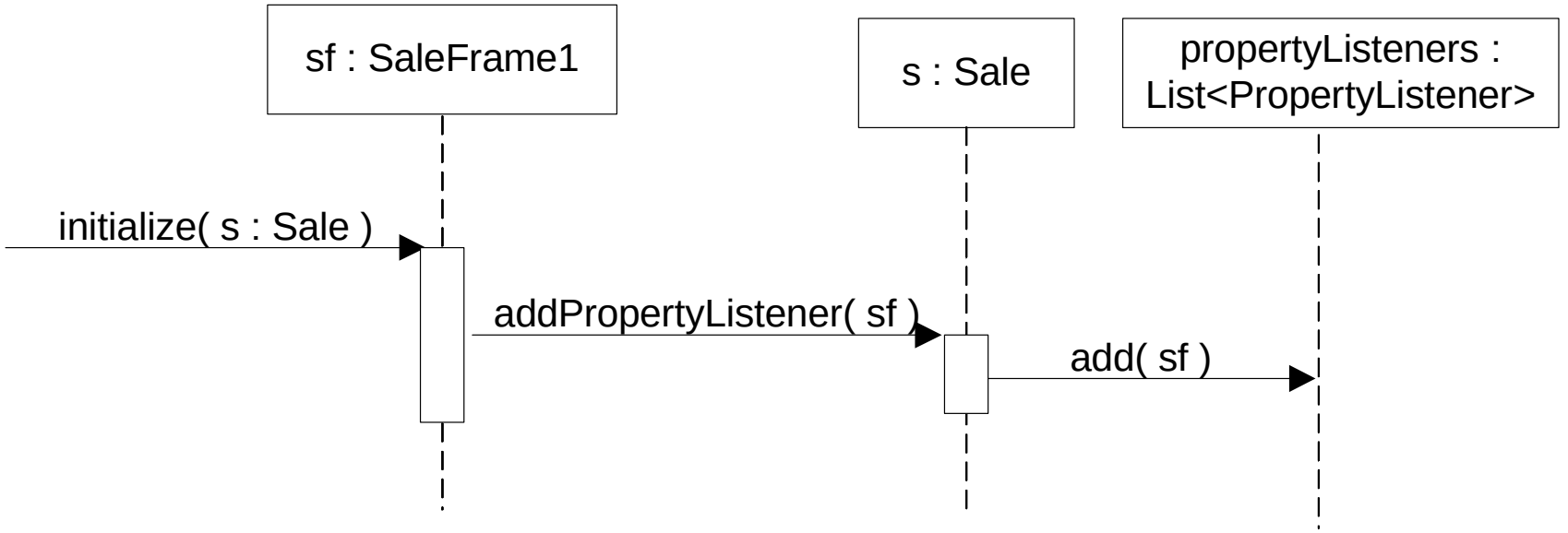
- Problem
  - Verschiedene Objekte (Observer) möchten über Zustandsänderungen eines Objekts (Observable) informiert werden
  - Direkte Kopplungen sind aber verboten
- Lösung
  - Listener Interface definieren, dass vom Observer implementiert wird und vom Observable über Änderungen benachrichtigt wird




{
for each PropertyListener pl in propertyListeners
pl.onPropertyEvent(this, name, value);
}
{
propertyListeners.add(lis);
}
Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
Publisher
{
total = newTotal;
publishPropertyEvent("sale.total", total);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
Einsatz von Interfaces !
Subscriber, Observer
{
if (name.equals("sale.total"))
saleTextField.setText(value.toString());
}
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
{
sale.addPropertyListener(this)
...
}

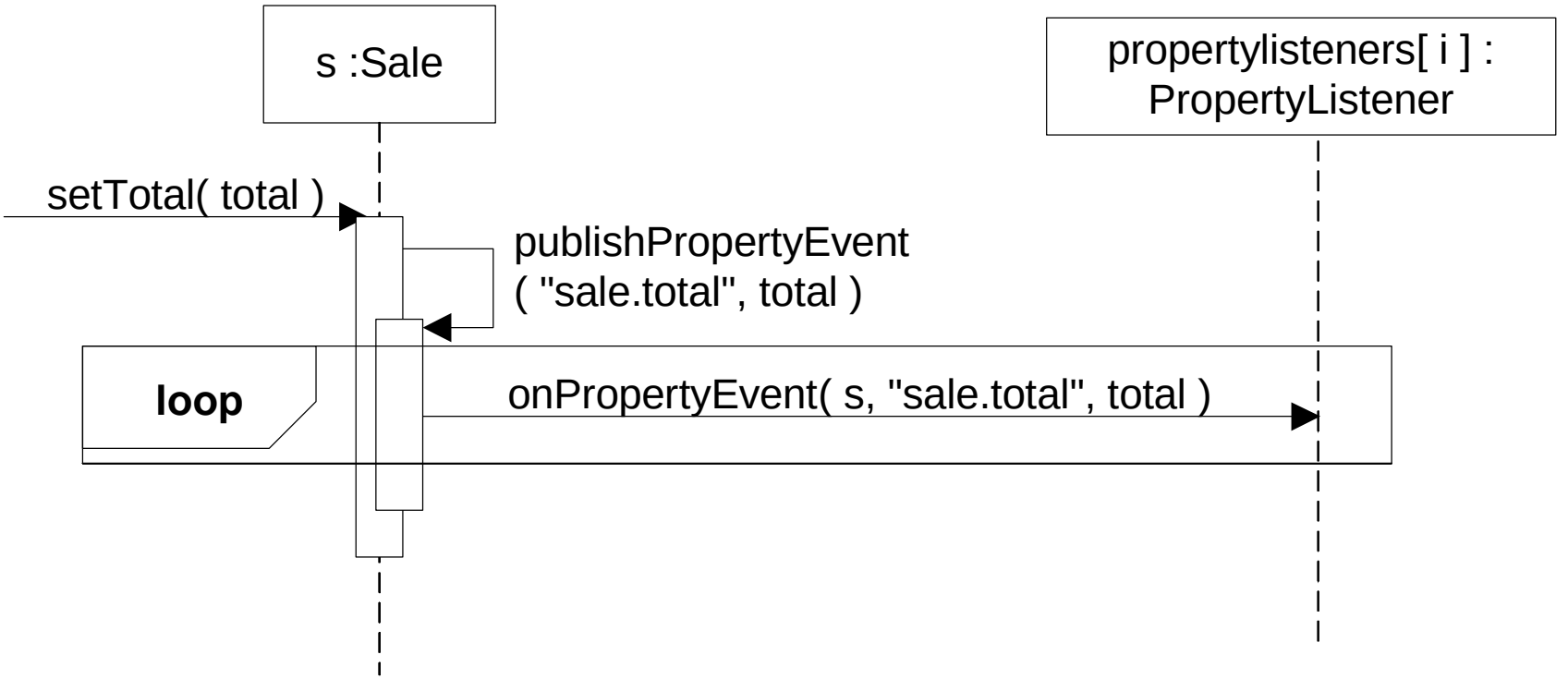
Initialisierung

- SaleFrame1 trägt sich als Observer von Sale ein
- Sale sieht in aber „nur“ als PropertyListener



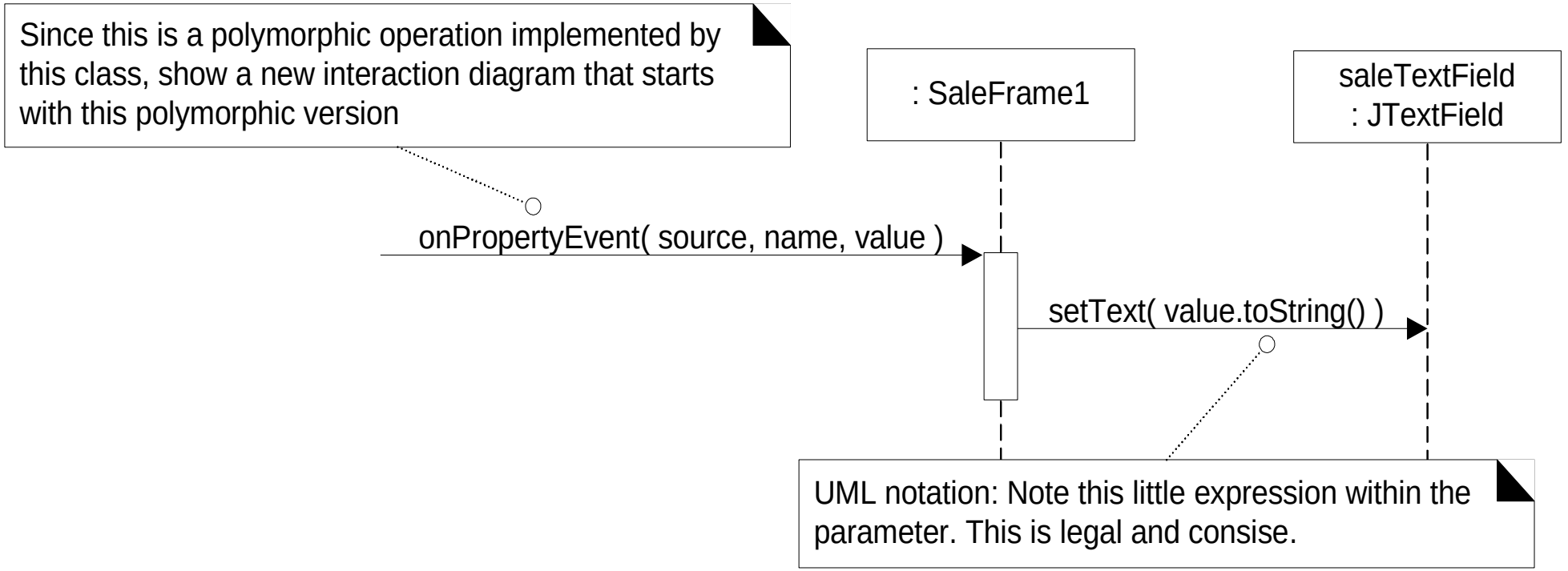
Auslösung Ereignis

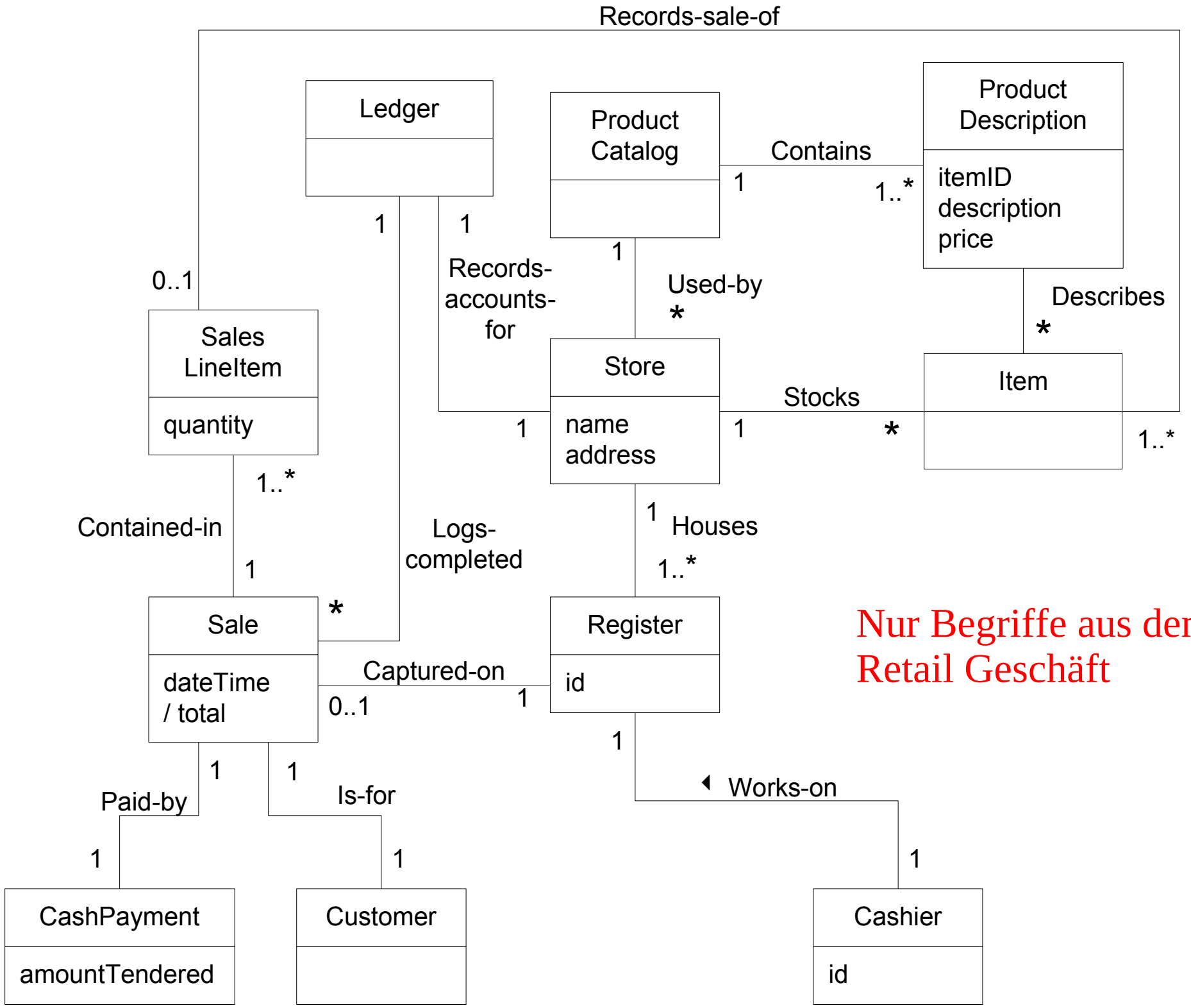
- Sale hat neues Total berechnet und veröffentlicht es
- Keine direkte Kopplung an SaleFrame1



Empfang Ereignis

- SaleFrame1 implementiert PropertyListener und erhält Ereignis





Nur Begriffe aus dem Retail Geschäft

Fragen

- Klasse „Database“ oder TcpIpConnection ?
  - Dürfen NIE in einem Domain Model auftauchen?
- Weshalb ist dieser „Low - Representational Gap“ so erstrebenswert?

Vorgehen

- Konzeptuelle Klassen finden
  - Wiederverwendung bestehender Modelle
  - Analysemuster einsetzen
  - Kategorienliste bei ziehen
  - Substantive aus Anwendungsfällen identifizieren
- UML Diagramm zeichnen
  - Klassendiagramm **ohne Operationen**
  - Nur **ungerichtete** Assoziationen
- Assoziationen und Attribute hinzufügen
  - Kategorienliste beiziehen

Kategorien Liste (Auszug)

- Geschäftstransaktionen
  - Transaktion als ganzes (Sale)
  - Transaktionsposition (SalesLineItem)
  - Produkt, das damit verbunden ist (Item)
  - Wo wird Transaktion registriert (Register)
  - Rollen von Personen, die beteiligt sind (Cashier)
  - Ort (Store)
- Beschreibung von Dingen
- Ereignisse mit Zeit/Ort
- Physische Objekte
- Kataloge

Kategorie Liste (2)

- Kataloge
- Container
- Dinge in Containern
- Andere beteiligte Systeme

Substantive herausziehen

- Substantive aus **Anwendungsfällen** herausziehen
- Kritisch überprüfen !
- Mehrdeutigkeit der natürlichen Sprache beachten
- Sich auf aktuell benötigte Konzepte konzentrieren

Richtlinien/Hinweise

- Verwaltung im Tool ?
  - vs. agile Modellierung
- Berichtsobjekte im Domänenmodell ?
  - Quittung : Nur wenn sie eine spezielle Rolle hat
- Fehler : Attribute anstelle von Klassen
  - Store (Name) als Attribut von Sale?
- Beschreibungsklassen einsetzen
  - ProductDescription

Beschreibungsklassen

Item
description price serial number itemID

Kann mit Datennormalisierung verglichen werden

Worse

ProductDescription
description price itemID

Describes

1 — *

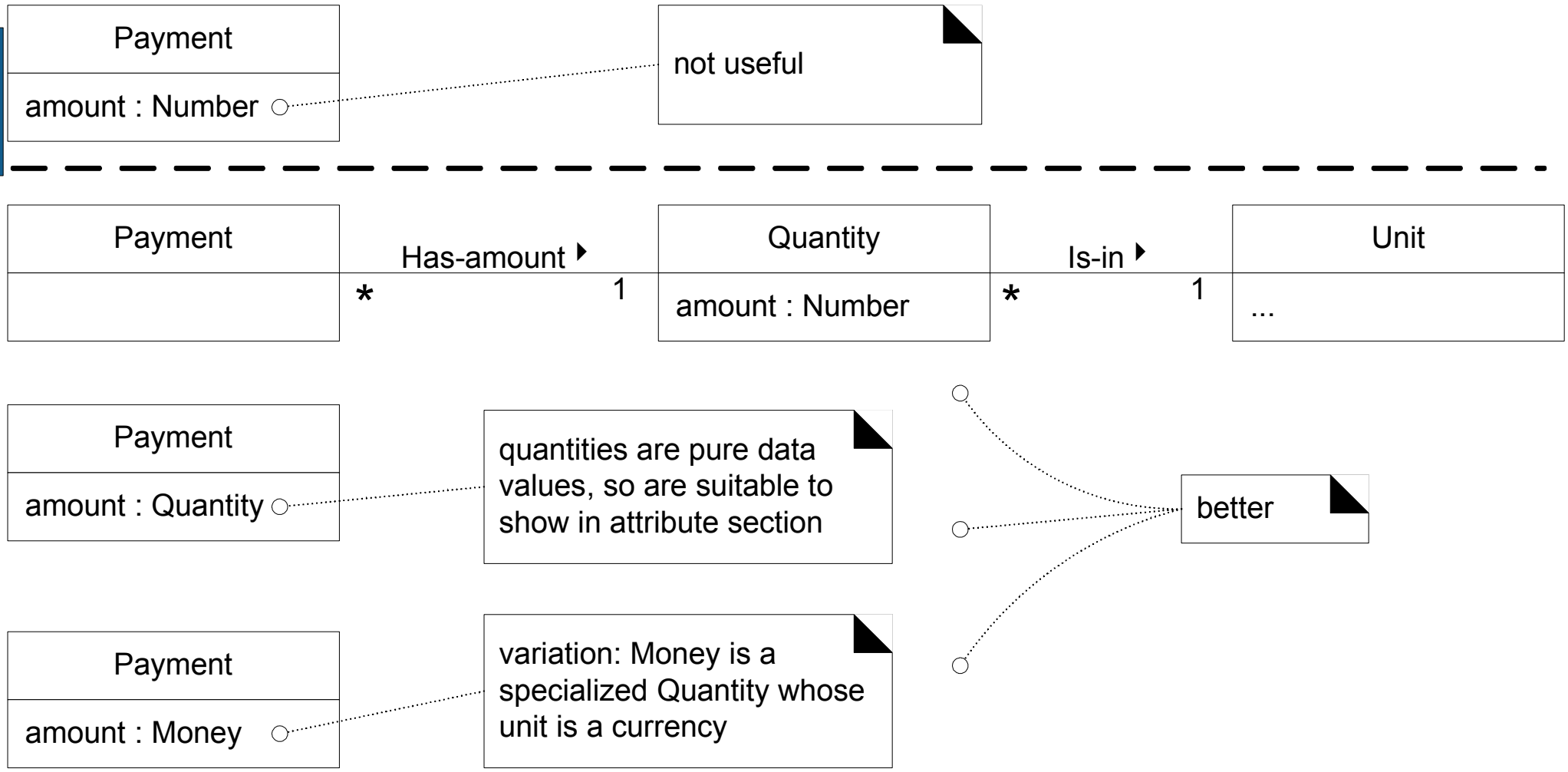
Item
serial number

Better

Richtlinien

- Keine komplexen Attribute, sondern eigene Klassen
- Attribute nicht als Fremdschlüssel modellieren !
- Mengen und Einheiten korrekt modellieren

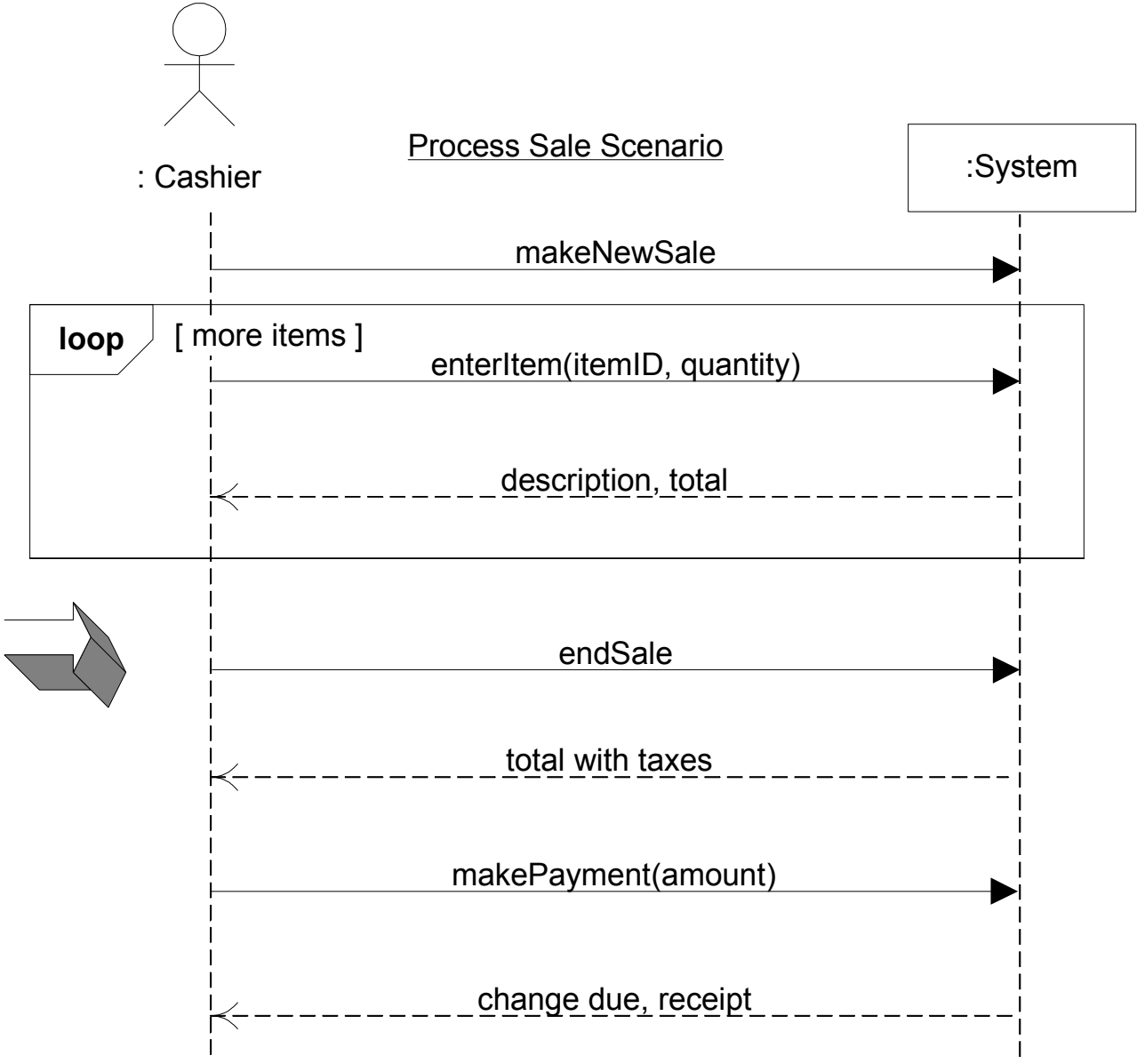
Mengen und Einheiten



Zusammenhang zu Use Case

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.
...



Operation Contract (Kap.11)

- Eine (System) Operation kann mit einem Vertrag noch genauer beschrieben werden
  - Vor- und Nachbedingungen
  - Nur beschreiben, was sich geändert hat, aber nicht wie
  - Basiert auf dem Domänenmodell
- Nachbedingung
  - Instanzen erstellt oder zerstört
  - Attribute geändert
  - Assoziationen hergestellt oder aufgehoben

Beispiel
Contract CO2 : enterItem

- Operation
  - enterItem(itemID : ItemID, quantity : integer)
- Querverweise
  - Use Case : Process Sale
- Vorbedingung
  - Ein Verkauf ist im Gange

Beispiel
Contract CO2 : enterItem

- Nachbedingung
  - SalesLineItem Instanz sli wurde erstellt
  - sli wurde mit dem aktuellen Sale verknüpft
  - sli.quantity wurde mit quantity initialisiert
  - Sli wurde über itemId mit einer ProductDescription verknüpft
- Klasse Item wurde bereits elliminiert, da die einzelnen Items nicht von Bedeutung sind.

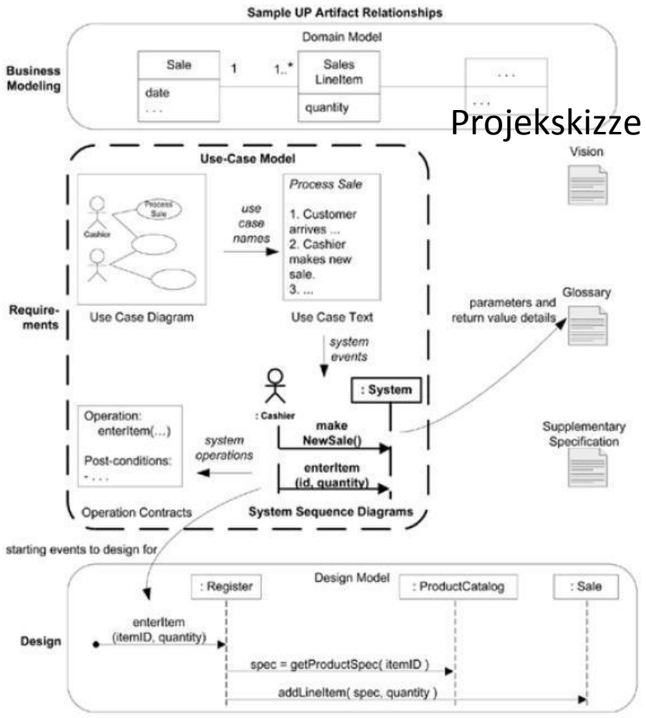
Concept Map SWEN1

Zurich University of Applied Sciences
zh aw School of Engineering

OO Analyse und Design (OOA/D)

(Requirements-)Analyse

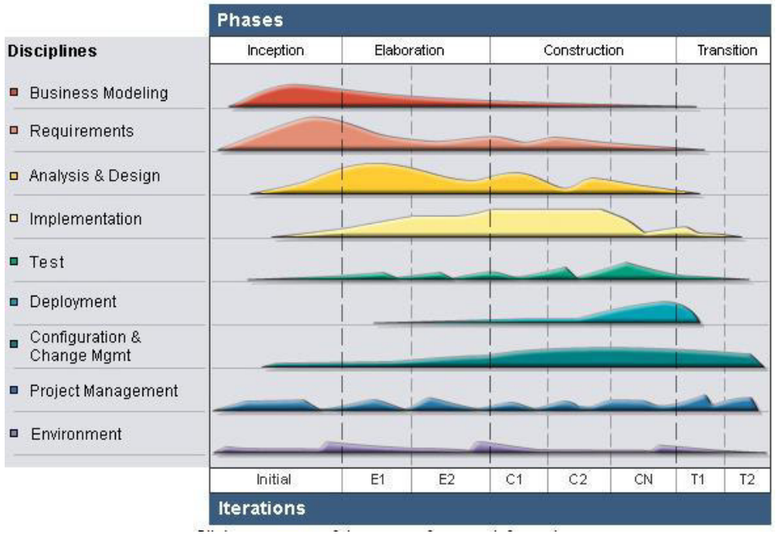
Software Requirements Specification (SRS)



Projekskizze

iterativ-inkrementeller und agiler Softwareentwicklungsprozess

Unified Process (UP)



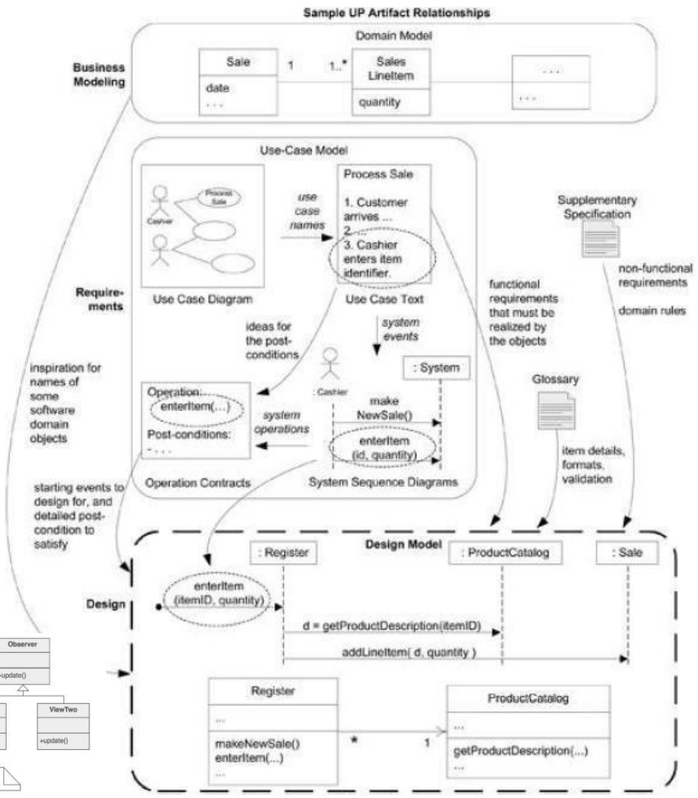
Artifacts Timing

Discipline	Artifact	Incep.	Elab.	Const.	Trans.
	Iteration⟶	I1	E1..En	C1..Cn	T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model (code, html, ...)		s	r	r

Modellierung mit der UML

Design

Responsibility Driven Design (RDD)

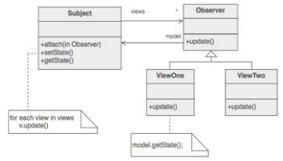


Design Patterns

GRASP, GoF

Refactoring

Clean Code



Projektmanagement

Software Development Plan

Overall Phase/Iteration Plan

Interation Plan(s) &
Interation Assessment(s)

Iteration #	Start	Ende	Mile-stone	Ziele, Release	Aufwand geplant [h]	Aufwand effektiv [h]
Inception Phase						
1	1.7.2015	30.7.2015	LCO	Vision, GUI Prototype	400	410
Elaboration Phase						
2	1.8.2015	31.8.2015		UC Zum Zielort navigieren (Standardablauf)	350	400
3	1.9.2015	30.9.2015	LCA	UC Gerät mit Smartphone koppeln (Standardablauf)	400	410
Construction Phase						
4	1.10.2015	30.10.2015		UC Power-on und Initialisierung, UC Zum Zielort navigieren	450	
5	1.11.2015	30.11.2015	IOC	UC Smartphone koppeln, UC Karten aktualisieren	400	
Transition Phase						
6	1.12.2015	23.12.2015	PR	Alpha-Release	200	

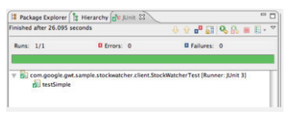
Risk List

Nr.	Name	Beschreibung	Wahrsch.	Schaden	Stufe	Prio	Massnahmen
1	Anbindung an bestehendes Buchhaltungssystem unklar	Ev. keine passenden Schnittstellen vorhanden	50%	Hoch	H	1	Schnittstellen abklären
2	Effizienz	muss mind. so effizient sein wie bestehende Kassen	30%	Sehr hoch	H	1	Resultierende Anforderungen analysieren

Testing

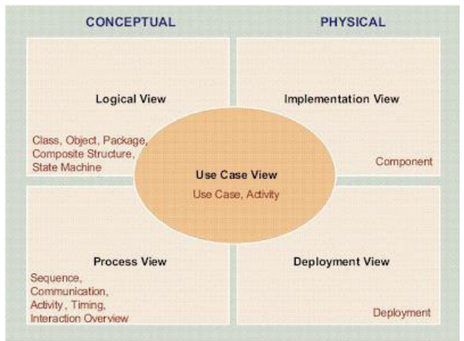
Teststufen, Testarten, Testplanung und Testdurchführung

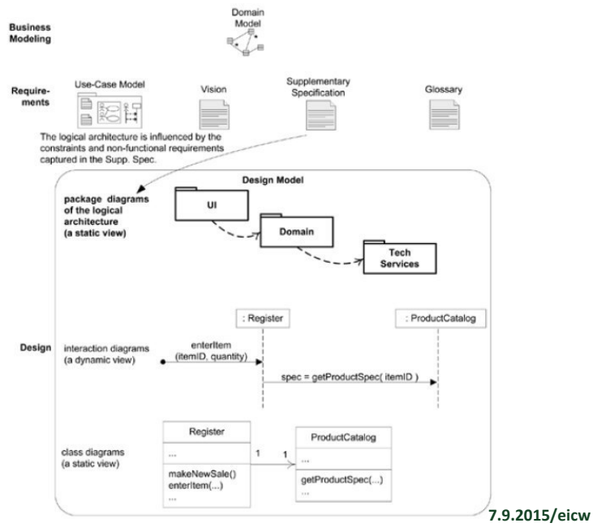
TDD, Unit Tests



Softwarearchitektur

Software Architecture Document (SAD)



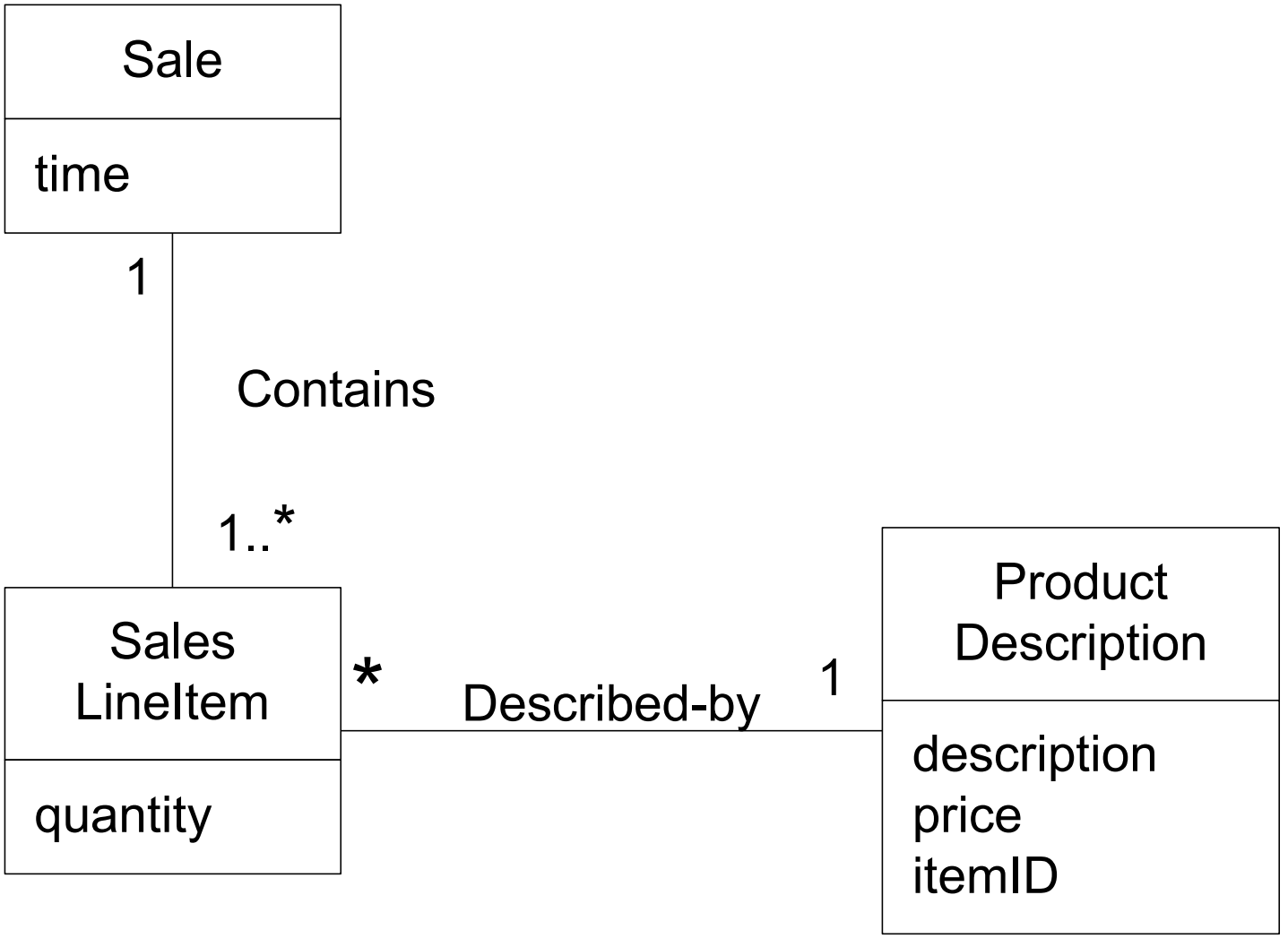


7.9.2015/eicw

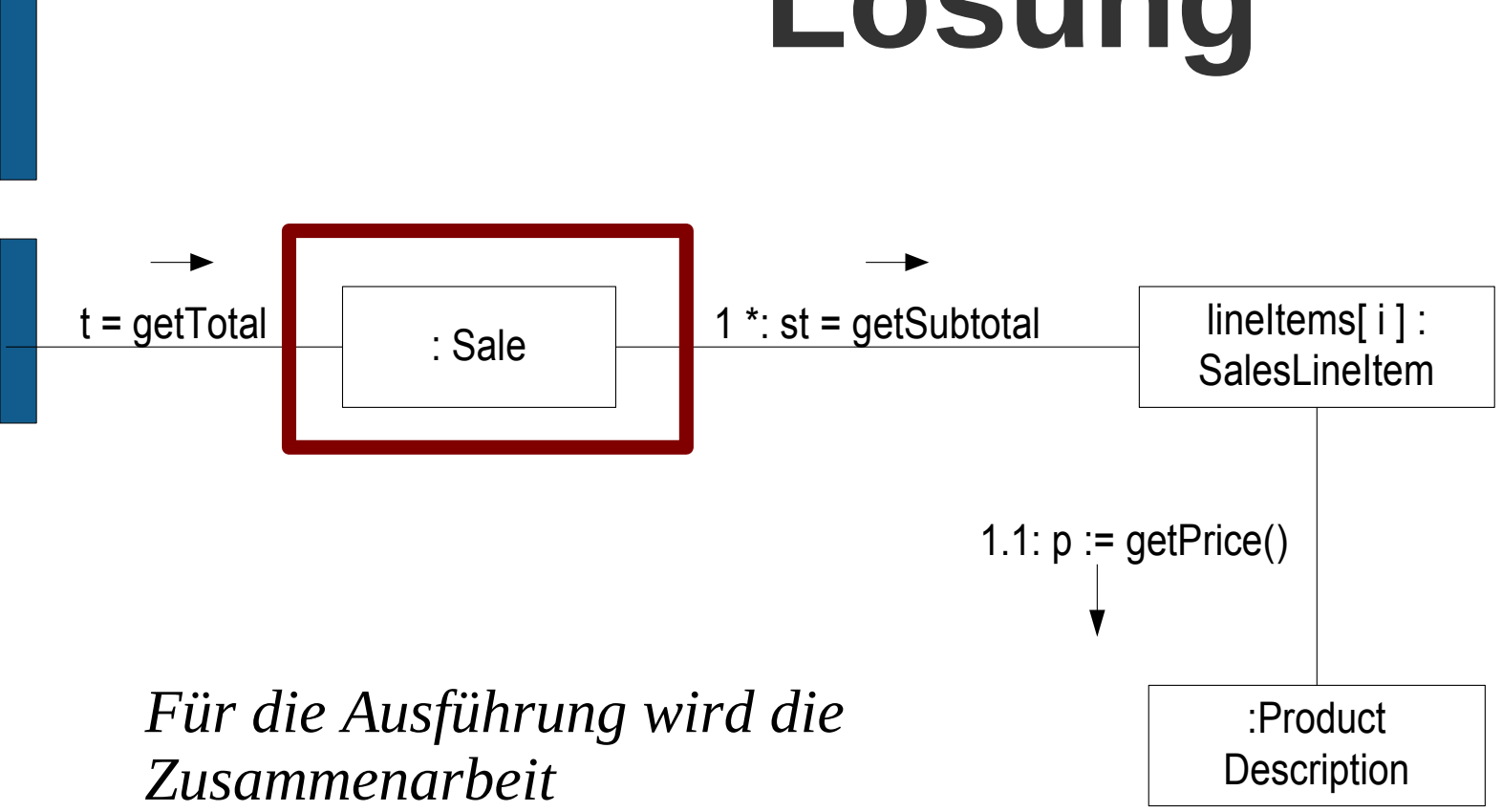
(1) Information Expert

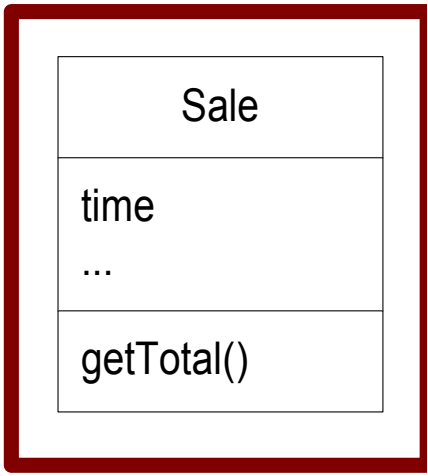
- Wichtigstes Prinzip der Zuteilung von Verantwortlichkeit
- Die Klasse, die die erforderlichen Informationen enthält, um die Verantwortlichkeit zu erfüllen
- Alternativen
  - Low Coupling / High Cohesion erfordern andere Lösung, nämlich eine „künstliche“ Klasse
- Auch partielle Verantwortlichkeiten möglich

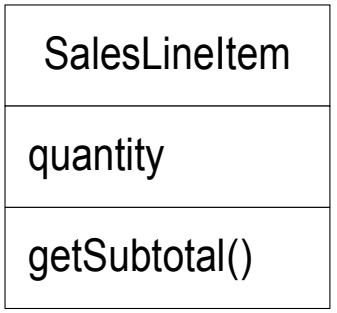
Wer berechnet Totalpreis eines Kaufs/Sale ?

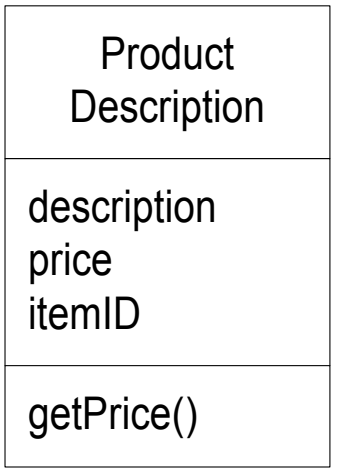


Lösung

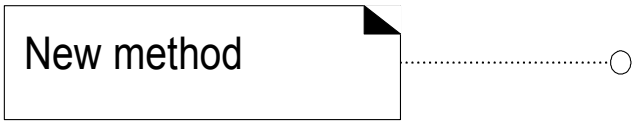








Für die Ausführung wird die Zusammenarbeit von anderen Klassen in Anspruch genommen



Was soll vermieden werden

- Reine Daten Klassen, die nur Behälter sind von Daten.
- SaleHandler oder SaleManager
  - Widerspricht Grundidee von OOP
  - Warum eine zweite Klasse einführen?
  - Information Hiding von Sale : Von aussen nur soviel Zugriff auf interne Attribute von Sale wie nötig. Bedingt, dass Methoden zur selben Klasse gehören wie private Attribute!
  - Wie kann Vererbung funktionieren?
- Register : Gesamter Code dort?

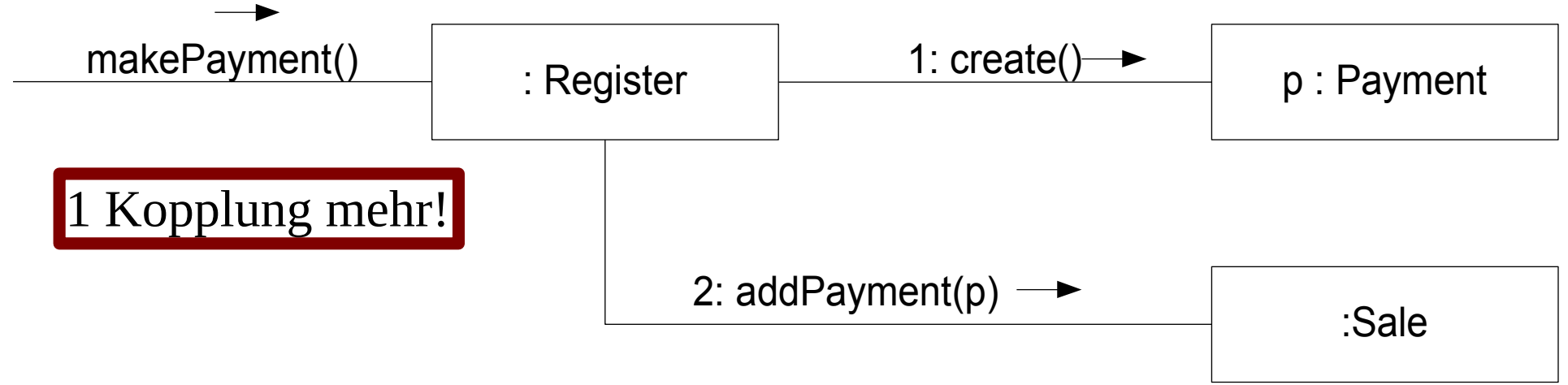
Oder anders gesagt

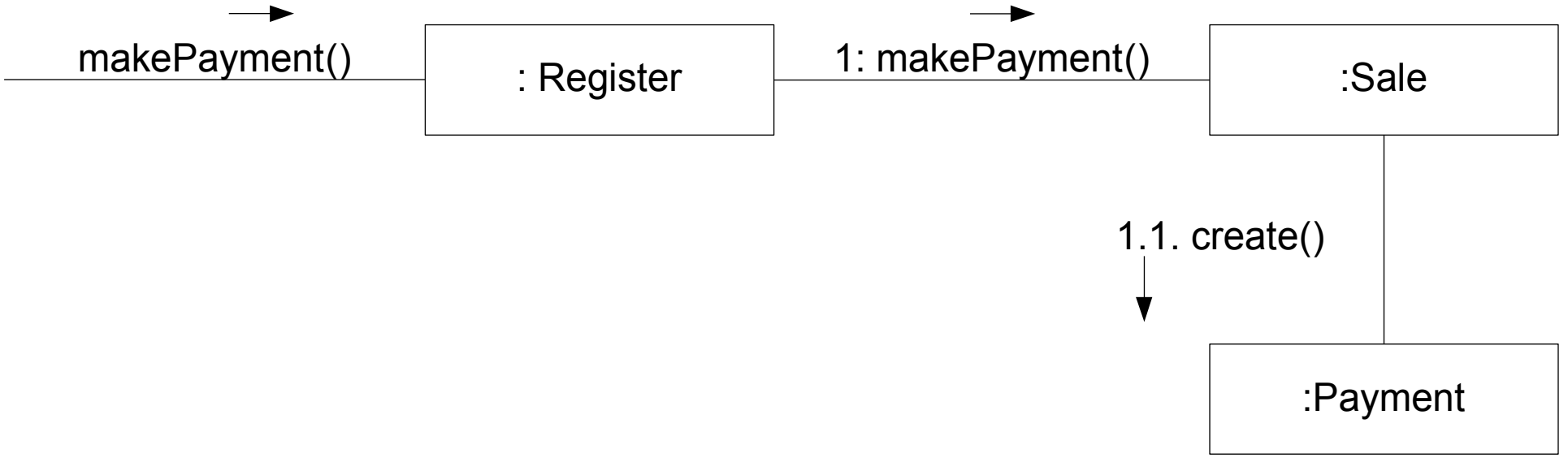
- „Do it myself“ Entwurfsmuster
  - Eine SW-Klasse macht das, was mit der entsprechenden Domänenklasse in der Realität gemacht wird.
- „Bring Objects To Life“
  - Wir versehen die SW Klassen mit Code und „animieren“ sie damit.
  - „Selbständige“ Objekte, die Daten und Methoden haben und somit komplexe Aufgaben lösen können.

(2) Low Coupling

- Wie können Änderungen möglichst lokal abgehandelt werden?
- Möglichst geringe Kopplung zwischen Klassen anstreben
- **Achtung** : Gewisse Kopplung ist aber notwendig für das Zusammenspiel der Klassen!
- Grundlegendes Prinzip!
- Qualitätskriterium für OO-Design

Wer erzeugt Payment?





(3) High Cohesion

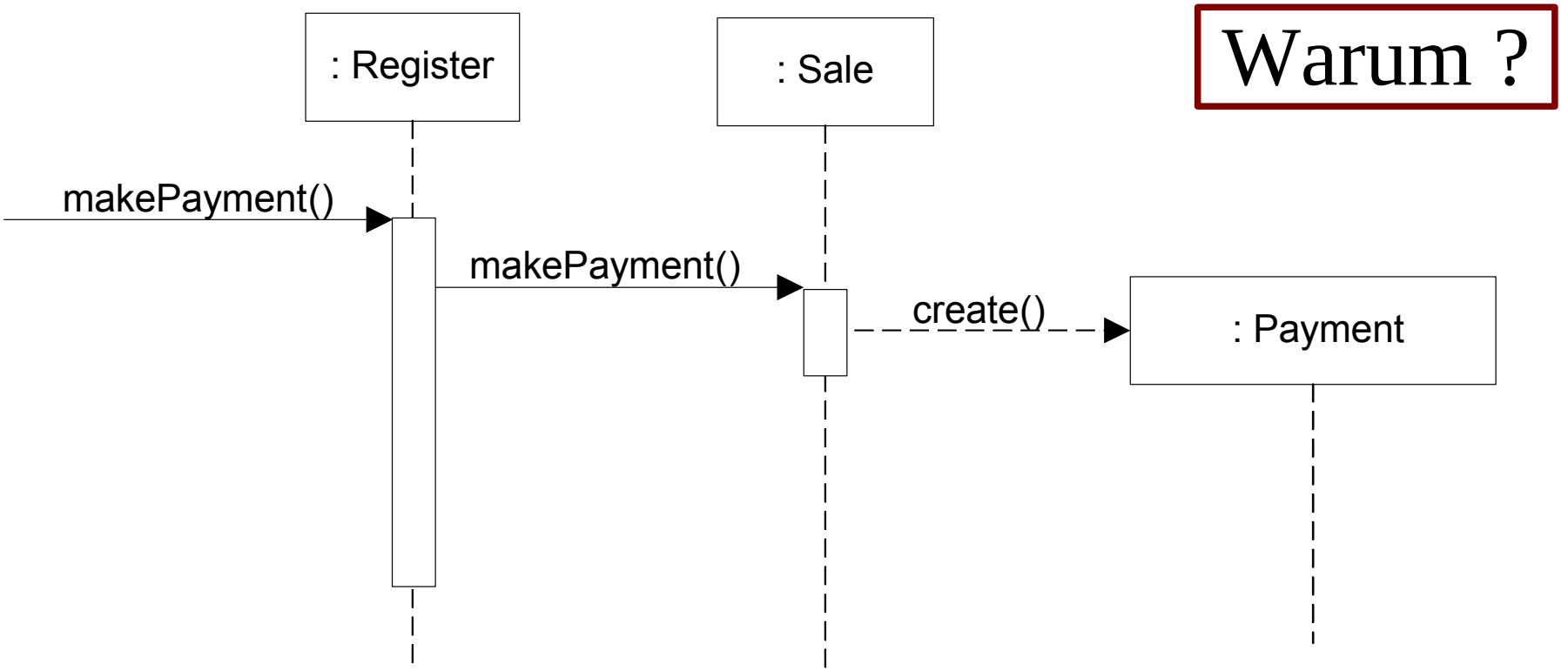
- Wie bleiben Objekte fokussiert und verständlich?
- Auf eine möglichst hohe innere Kohäsion achten
  - Modularer Entwurf
  - Ergänzt sich gut mit Low Coupling
- Wenige Ausnahmen denkbar
  - Einfachere Wartung : SQL Code in einer Klasse zentralisieren
- Klare, eindeutige und schmale Verantwortlichkeit

Beispiele für verschiedene Stufen

- Sehr gering
  - 1 Klasse für RDB und RPC zuständig
- Gering
  - 1 Klasse für gesamte RDB zuständig
- Hoch
  - Viele Klassen, die RDB Zugriff ermöglichen, 1 z.B nur für Resultat zuständig
- Moderat
  - Fasst verschiedene Bereiche zusammen, ohne aber selber etwas zu machen

Wer erzeugt Payment?

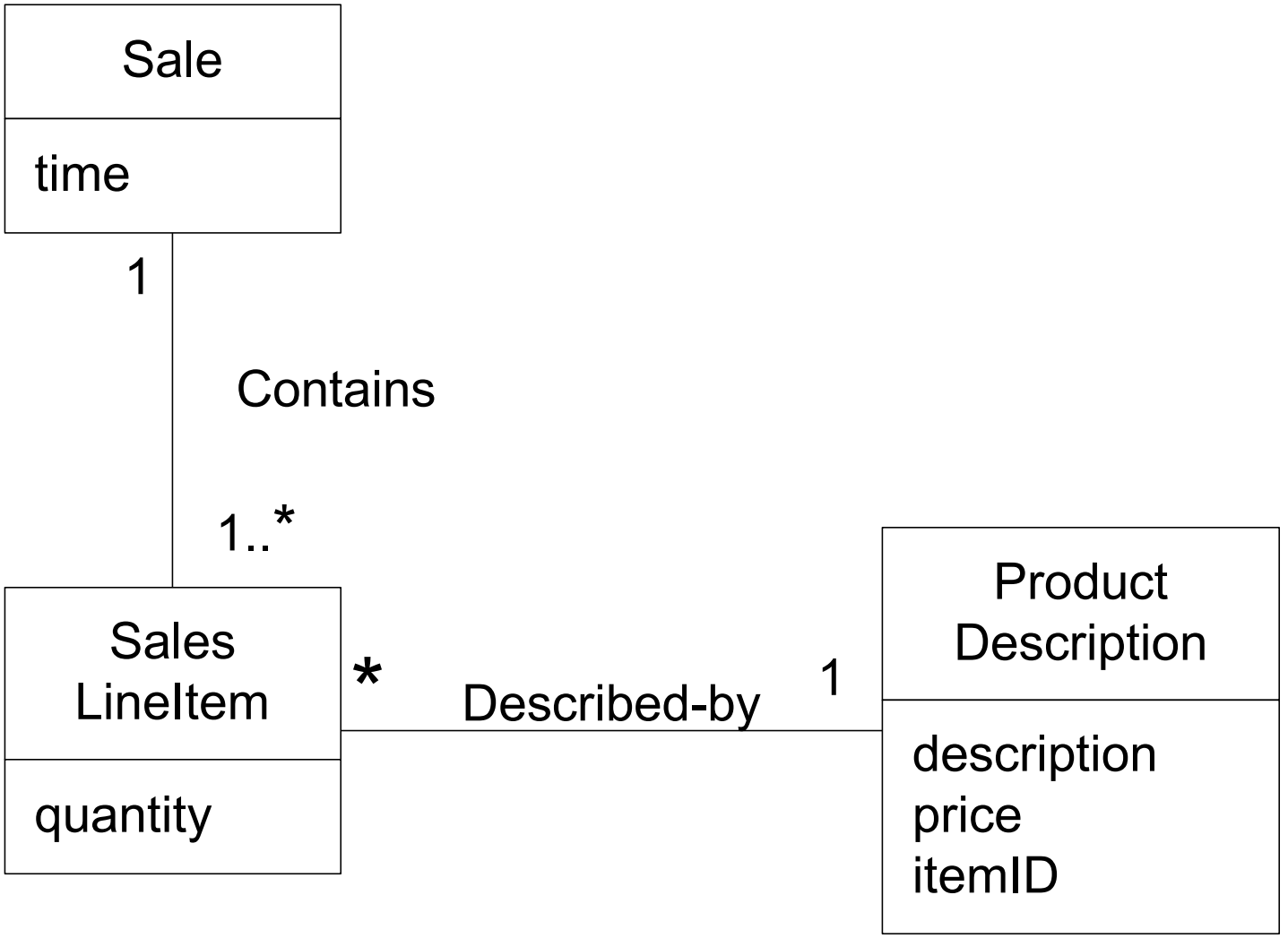
- High Cohesion rät von Register ab und bevorzugt Sale, wie Low Coupling



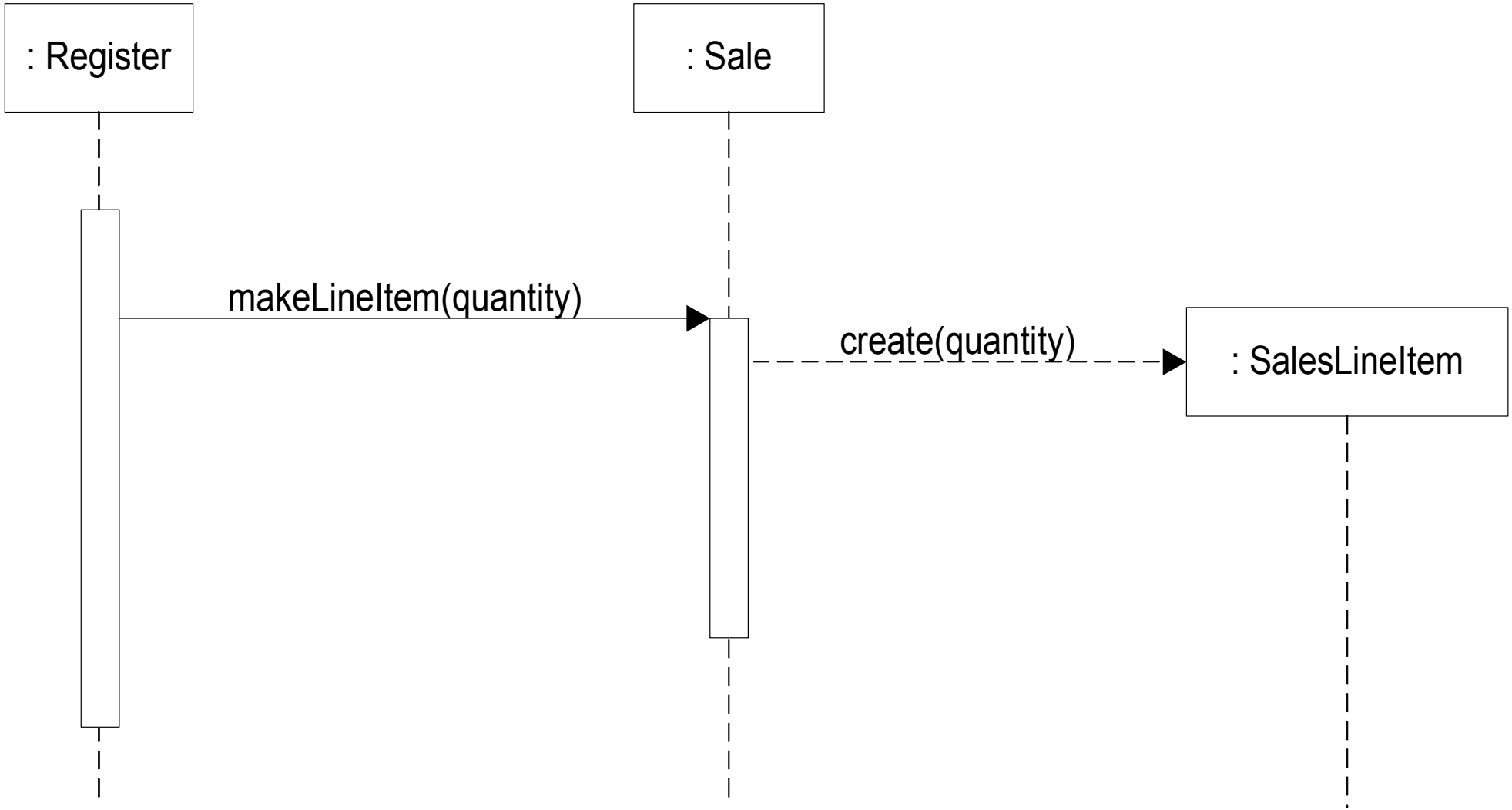
(4) Creator

- Wer erstellt Instanzen einer bestimmten Klasse?
- Kandidaten für diese Verantwortlichkeit
  - Behälterklasse
  - Klasse mit enger Zusammenarbeit
  - Klasse, die Initialisierungsdaten kennt
- Alternativen
  - Factory Klasse
- Unterstützt geringe Kopplung

Wer erzeugt SalesLineItems ?



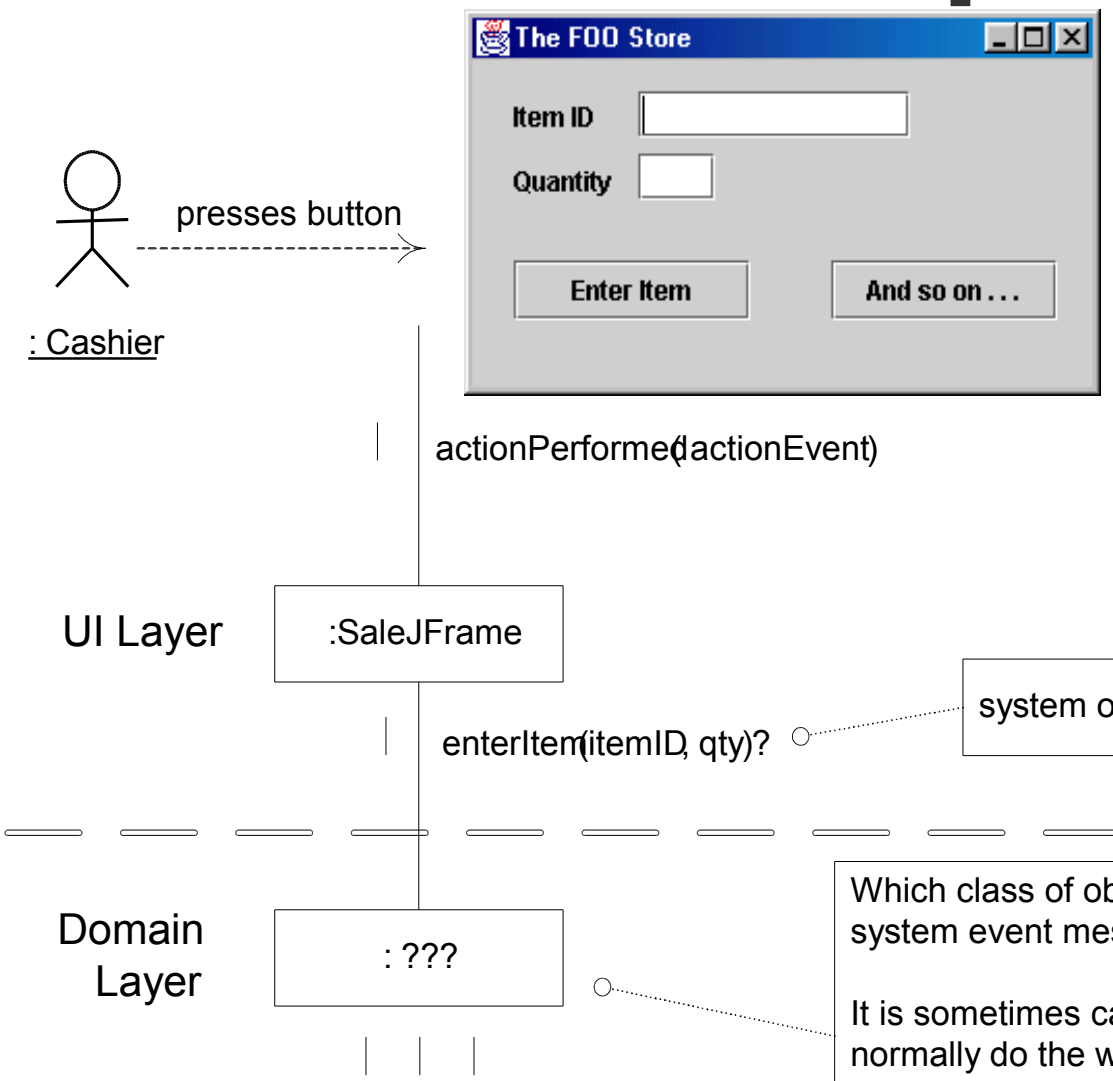
Lösung


: Register
: Sale
makeLineItem(quantity)
create(quantity)
: SalesLineItem

(5) Controller

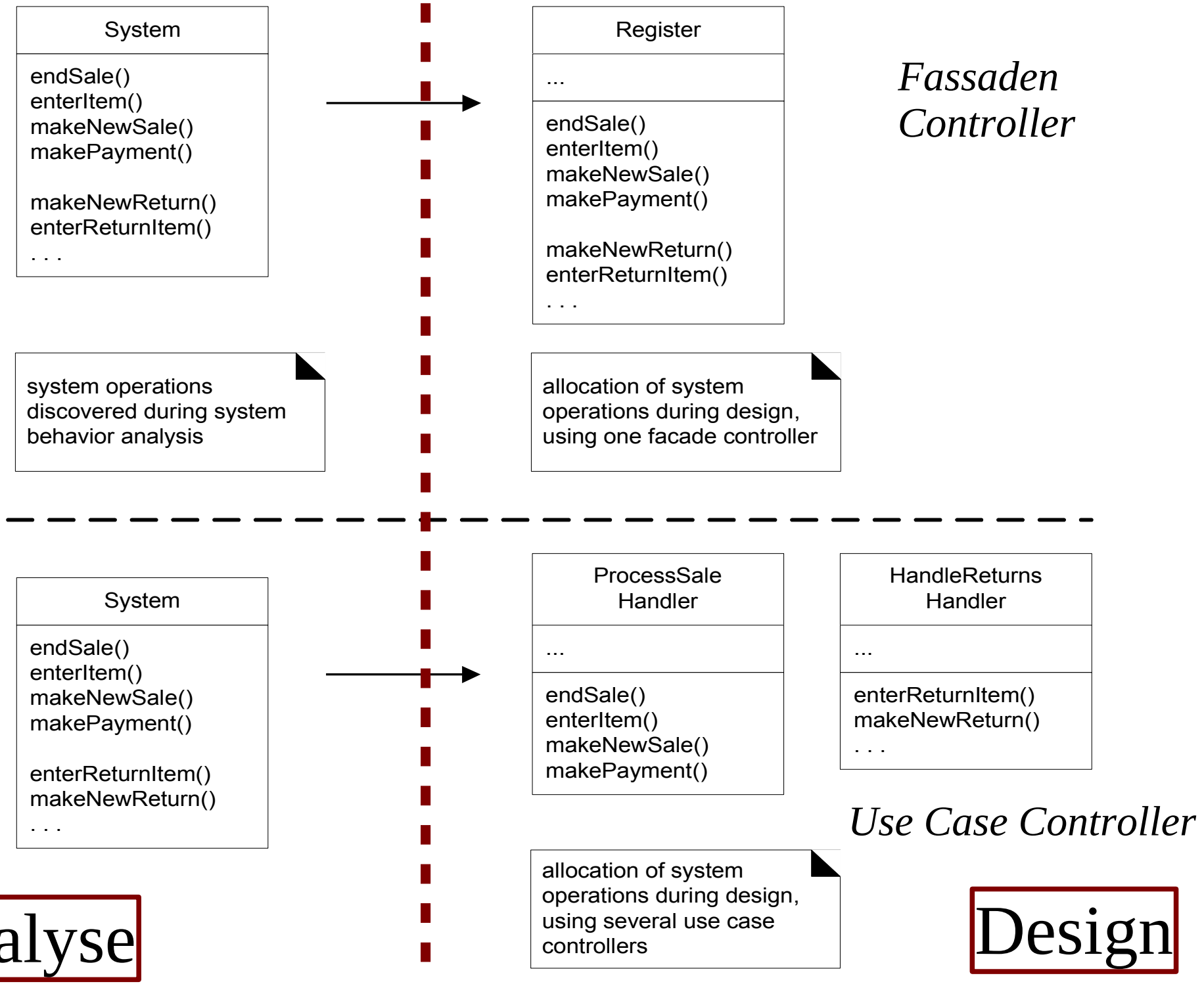
- Welches Objekt realisiert (in erster Instanz) die Systemoperationen ?
- A) Fassaden Controller
  - „Root“ Objekt, System, übergeordnetes System
- B) Use-Case Controller
  - Pro Anwendungsfall eine „künstliche“ Klasse
- Wichtig : Controller macht selber nur wenig und delegiert fast alles!

Beispiel

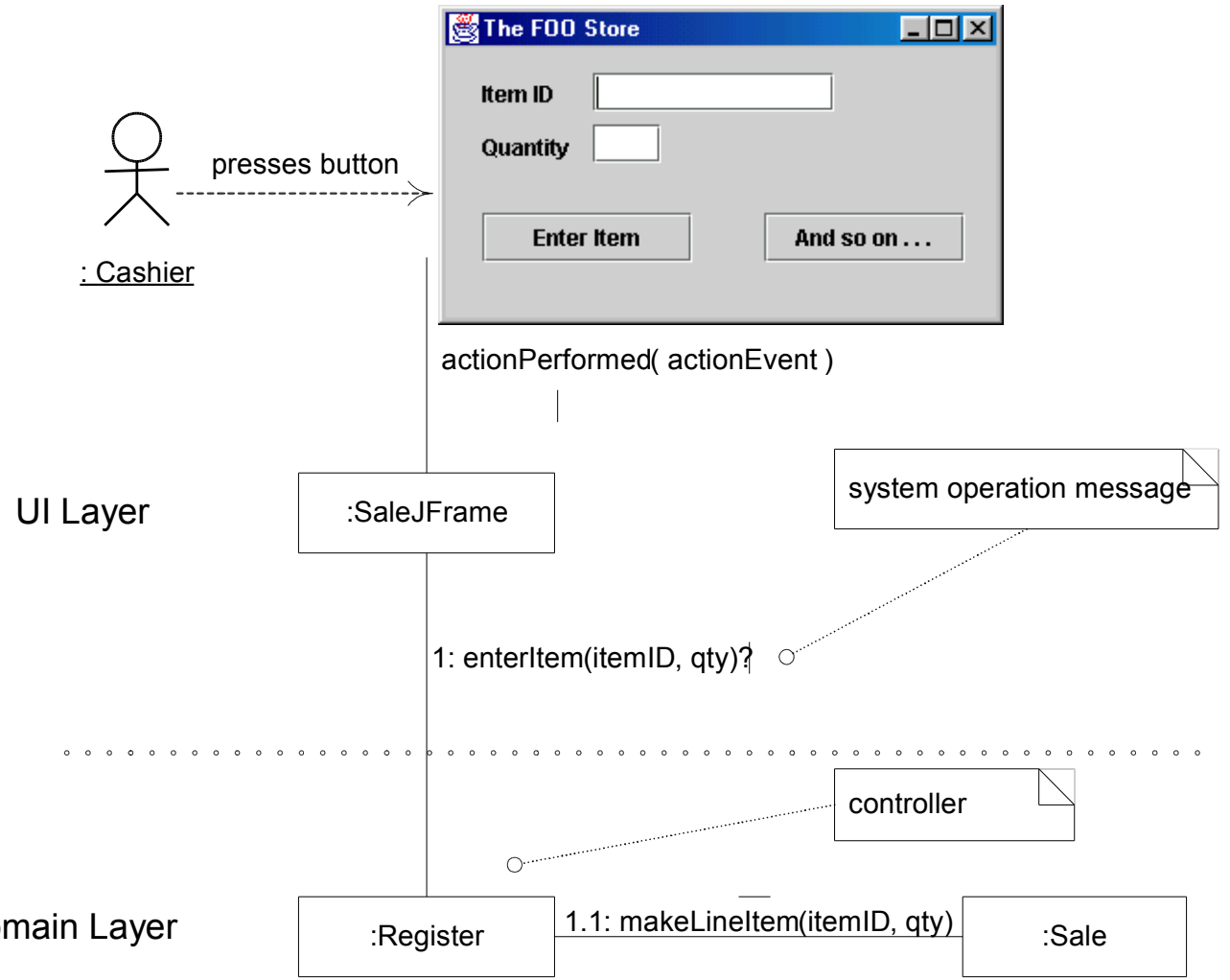

The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)?
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects
The controller is a kind of "facade" onto the domain layer from the interface layer.

Hinweise

- Controller ist „oberstes“ Objekt der Domänenlogik an der Schnittstelle zur nächst höheren Schicht
- „Erste Ansprechsperson“
- UI selber hat keine Domänenlogik und bearbeitet die Systemereignisse nicht selber
- Die erste Entwurfsfrage, die beantwortet werden muss !


System
endSale()
enterItem()
makeNewSale()
makePayment()
makeNewReturn()
enterReturnItem()
. . .
Register
...
endSale()
enterItem()
makeNewSale()
makePayment()
makeNewReturn()
enterReturnItem()
. . .
Fassaden Controller
system operations discovered during system behavior analysis
allocation of system operations during design, using one facade controller
System
endSale()
enterItem()
makeNewSale()
makePayment()
enterReturnItem()
makeNewReturn()
. . .
ProcessSale Handler
...
endSale()
enterItem()
makeNewSale()
makePayment()
HandleReturns Handler
...
enterReturnItem()
makeNewReturn()
. . .
Use Case Controller
allocation of system operations during design, using several use case controllers
Analyse
Design

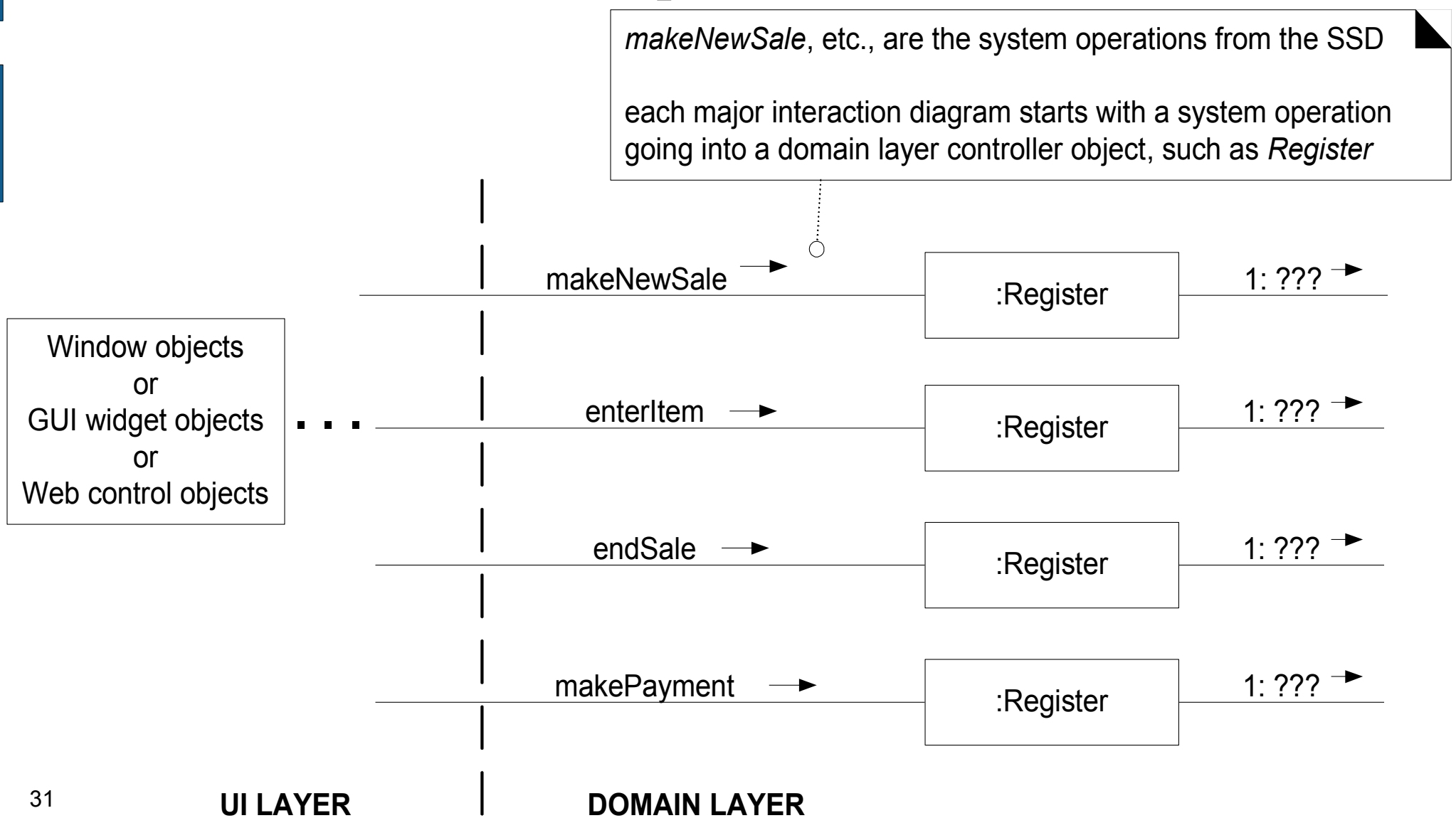
Umsetzung in Fallstudie



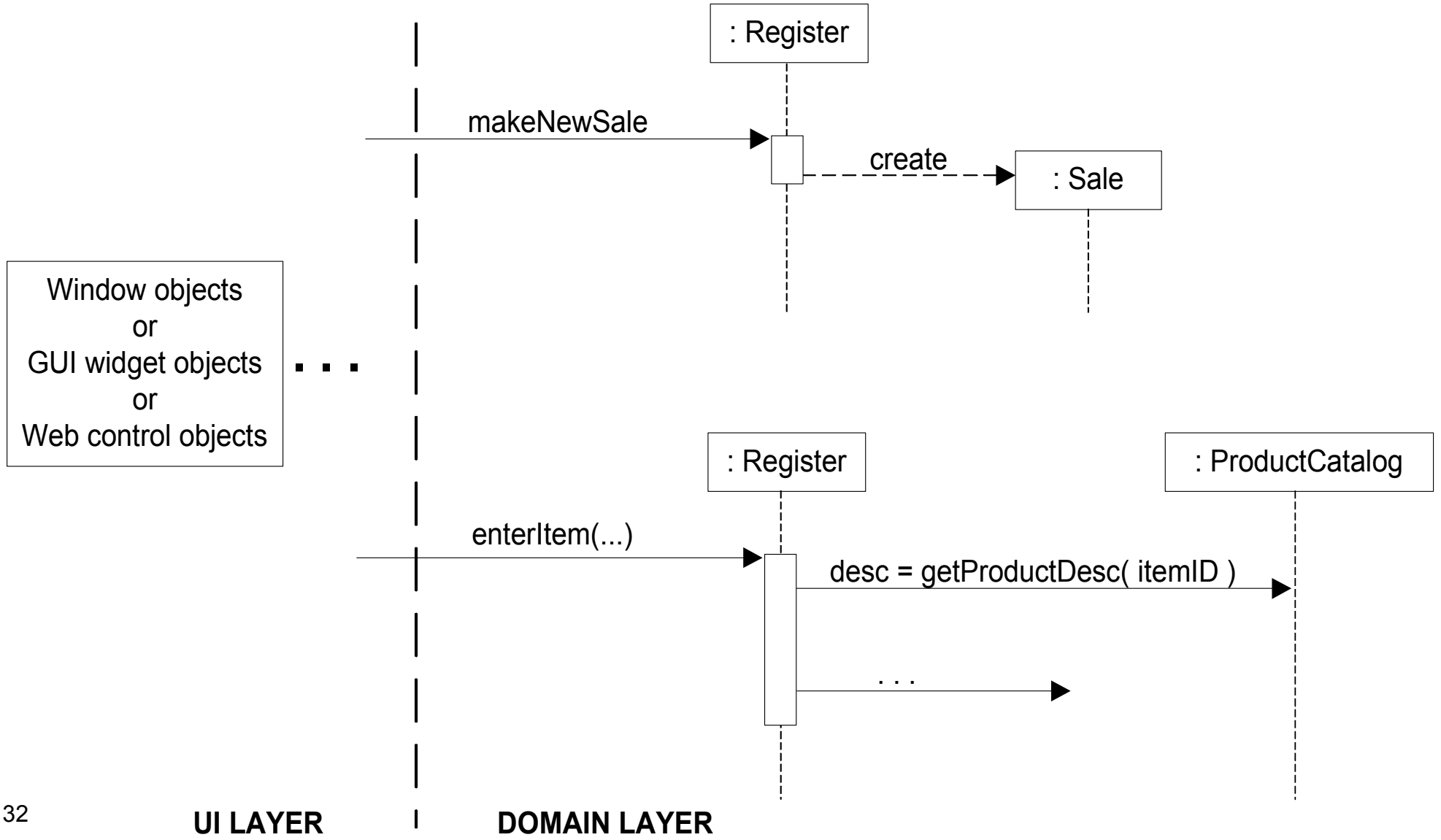
Was bedeutet „Use Case Realisation“ ?

- Realisierung eines Anwendungsfalls mit kollaborierenden Objekten
- Baut auf :
  - Anwendungsfälle
  - System Sequenzdiagramm (SSD)
  - Operation Contracts
- Lösung wird mit Interaktionsdiagrammen veranschaulicht
- Eigentliche Designtätigkeit !

Systemoperation ist Startpunkt ...



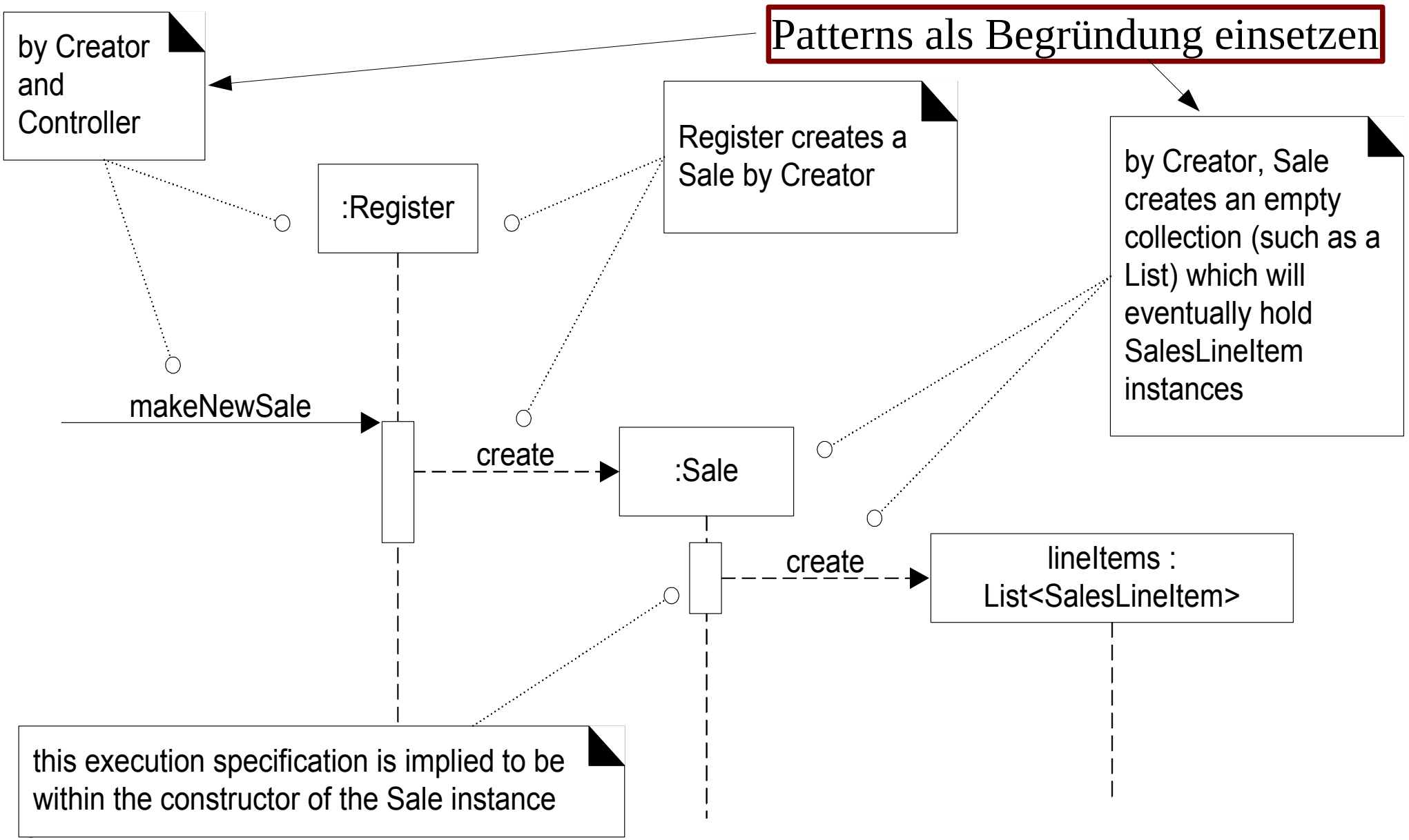
… „Rechte Seite“, das „Innere“ entwerfen



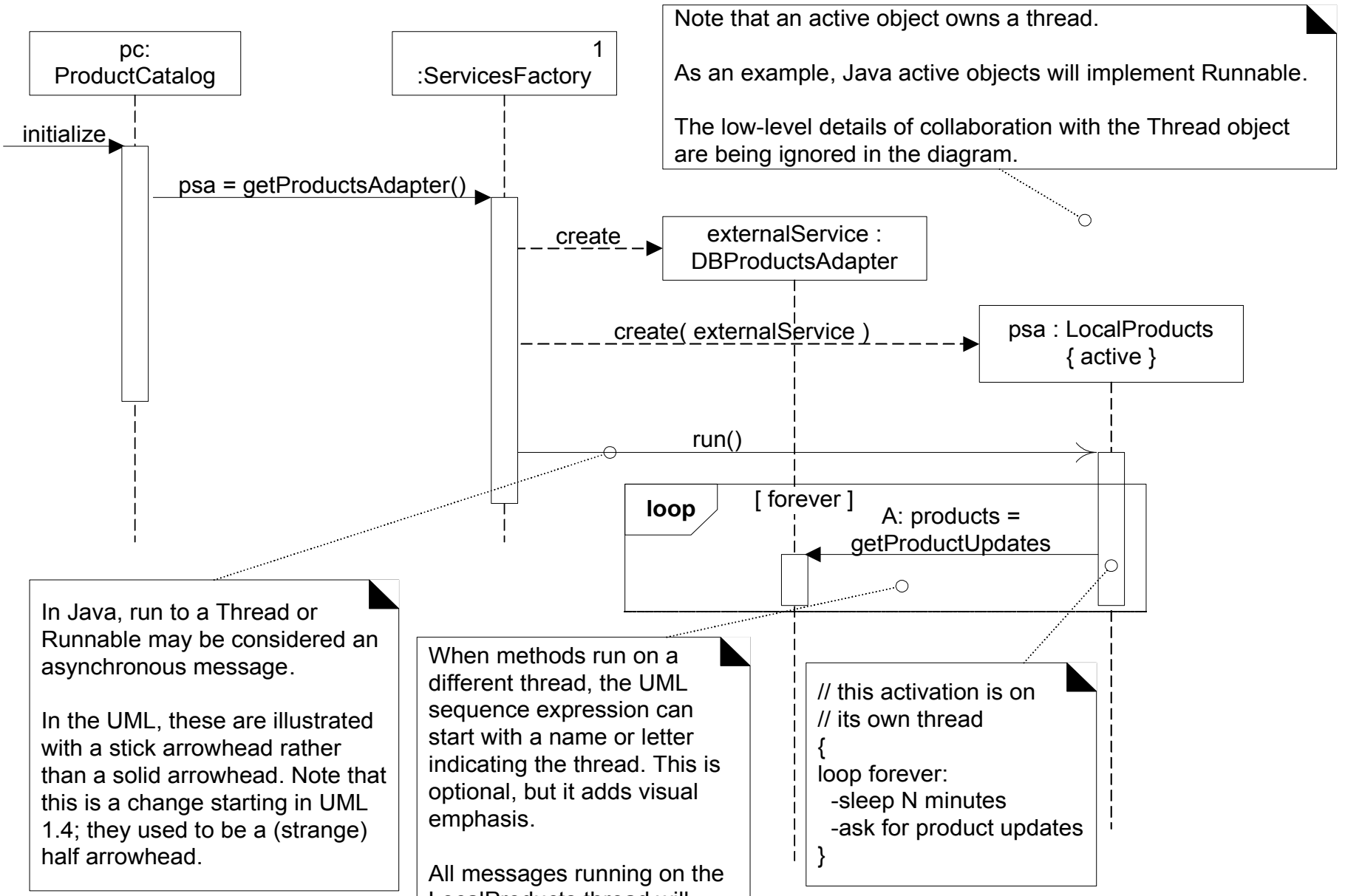
Beispiel „MakeNewSale“

- MakeNewSale
- Input
  - Operation Contract
    - Instanz s von Sale wurde erstellt
    - S wurde mit Register verbunden
    - Attribute von S wurden initialisiert
  - Anwendungsfall
- Lösung mit Hilfe von Entwurfsmustern entwickeln (kreativer und systematischer Prozess)

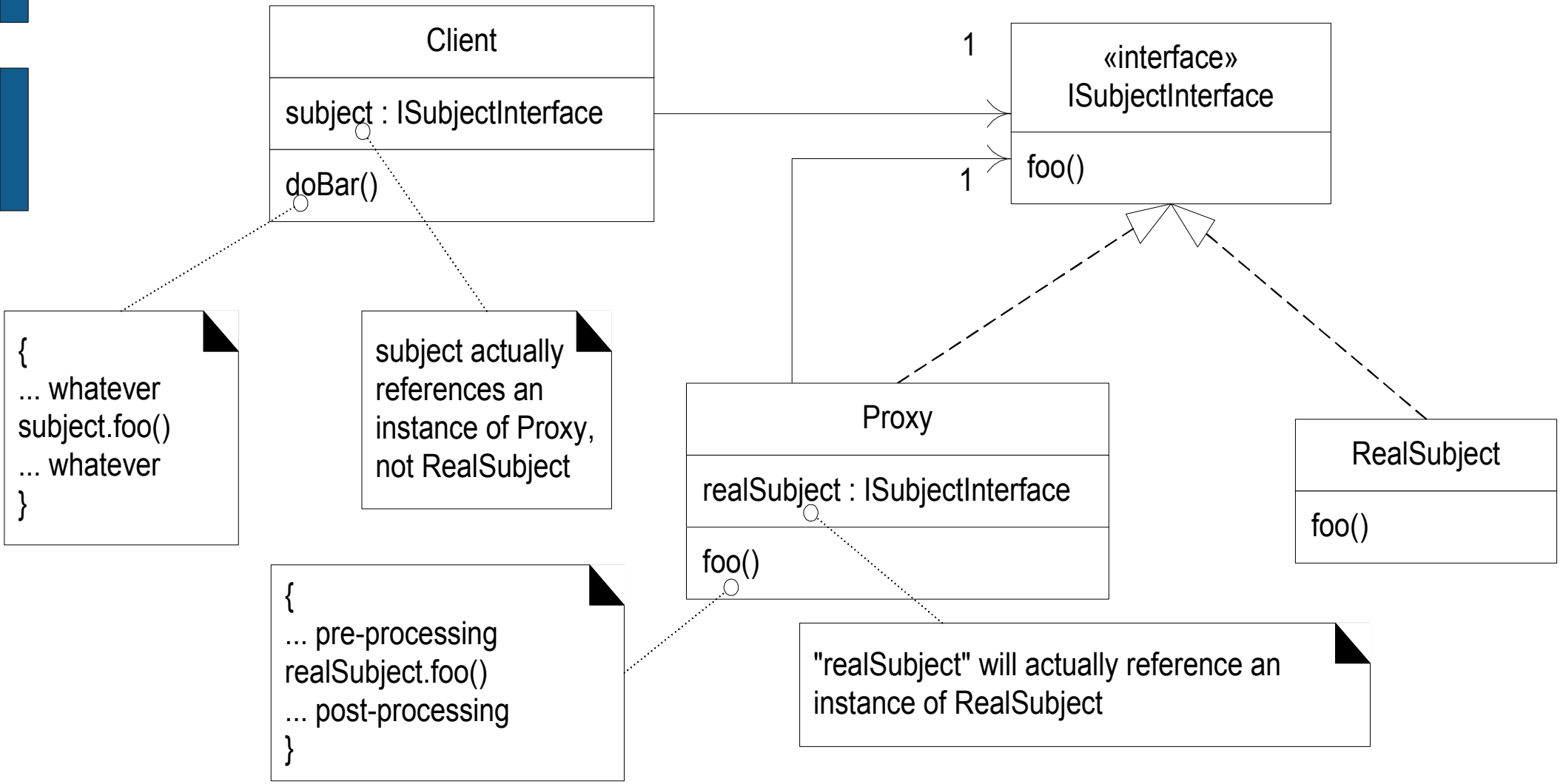
... Lösung

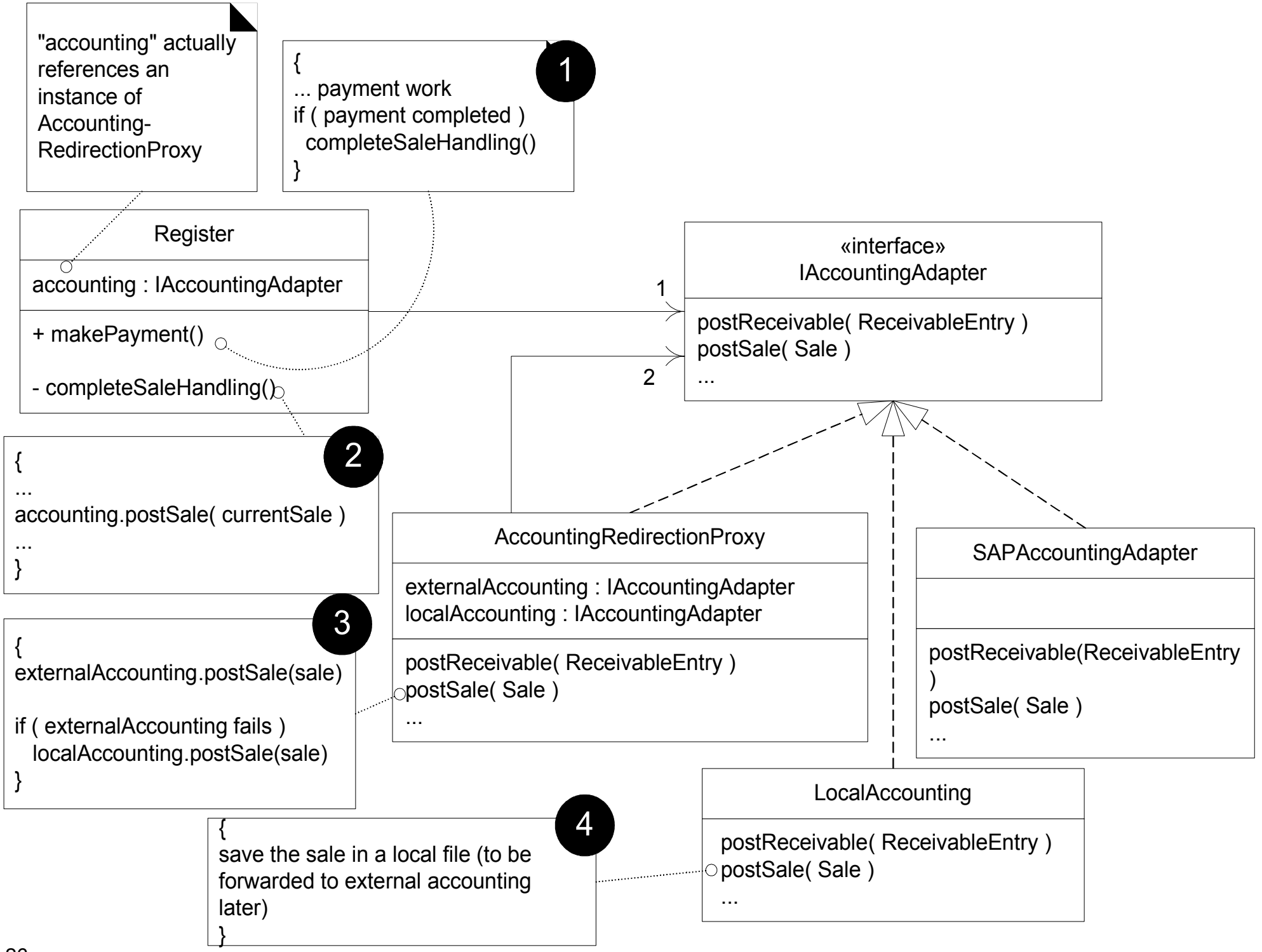

Patterns als Begründung einsetzen
by Creator and Controller
Register creates a Sale by Creator
by Creator, Sale creates an empty collection (such as a List) which will eventually hold SalesLineItem instances
:Register
makeNewSale
create
:Sale
create
lineItems : List<SalesLineItem>
this execution specification is implied to be within the constructor of the Sale instance

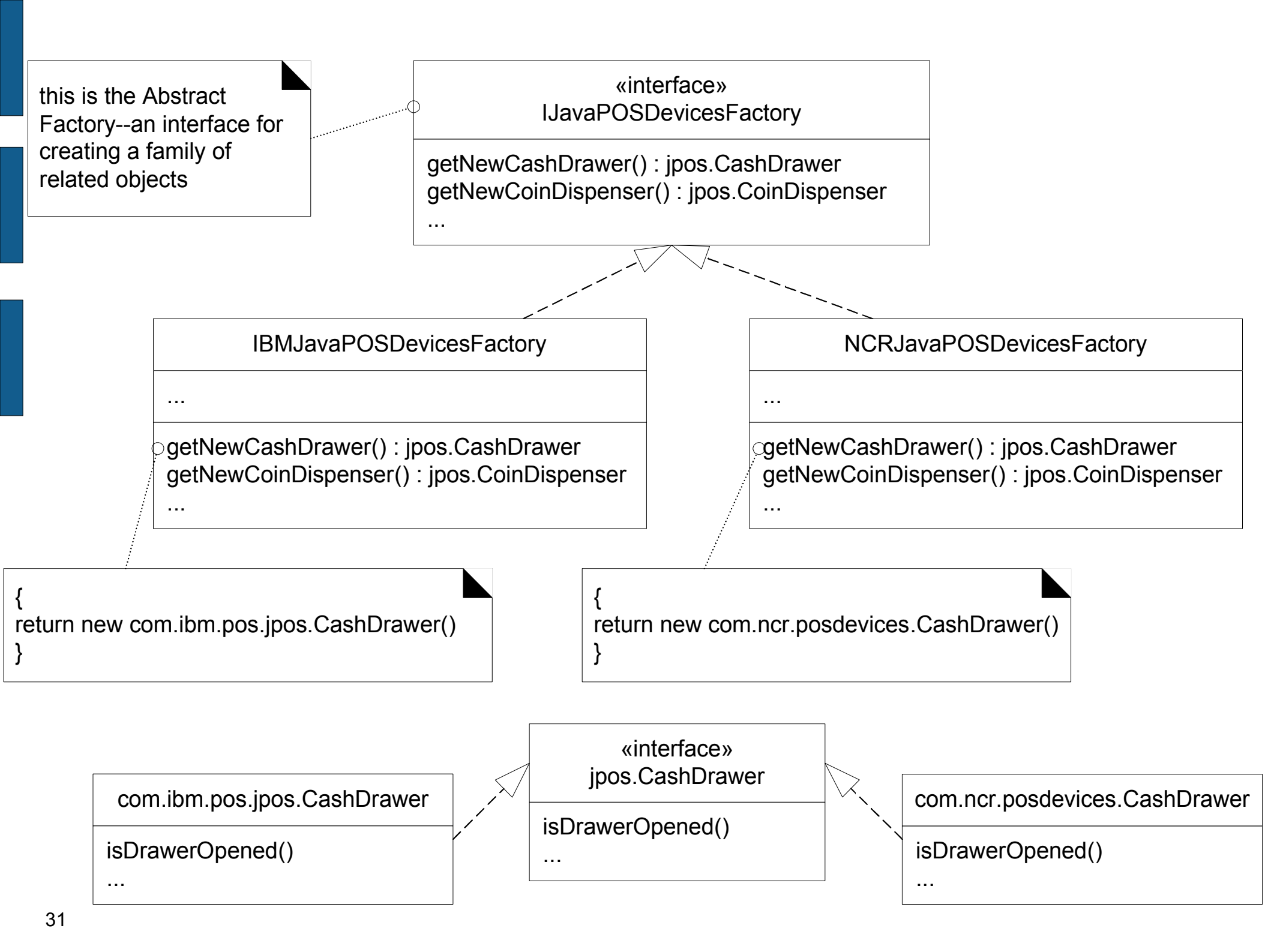
Threads in UML

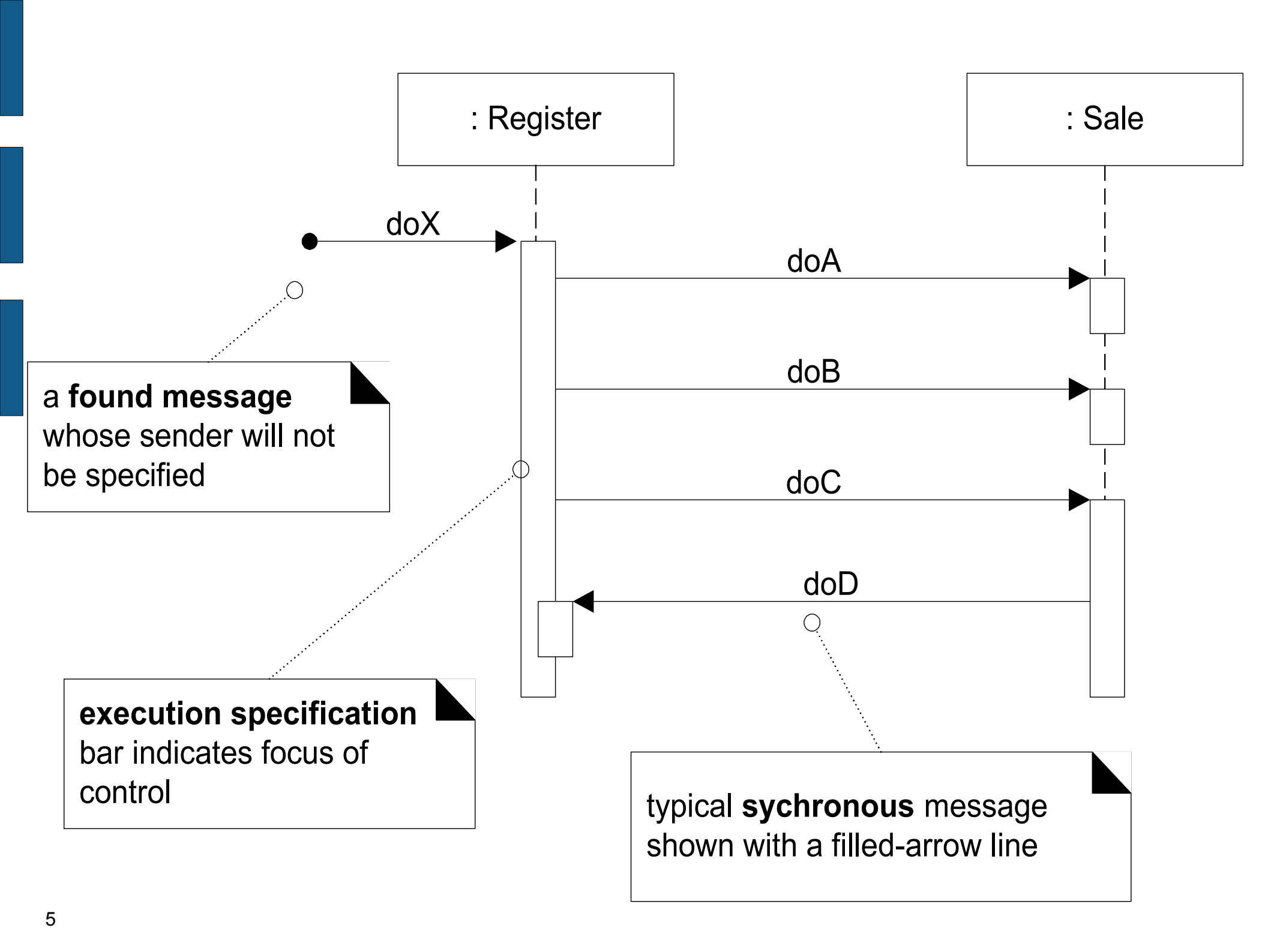


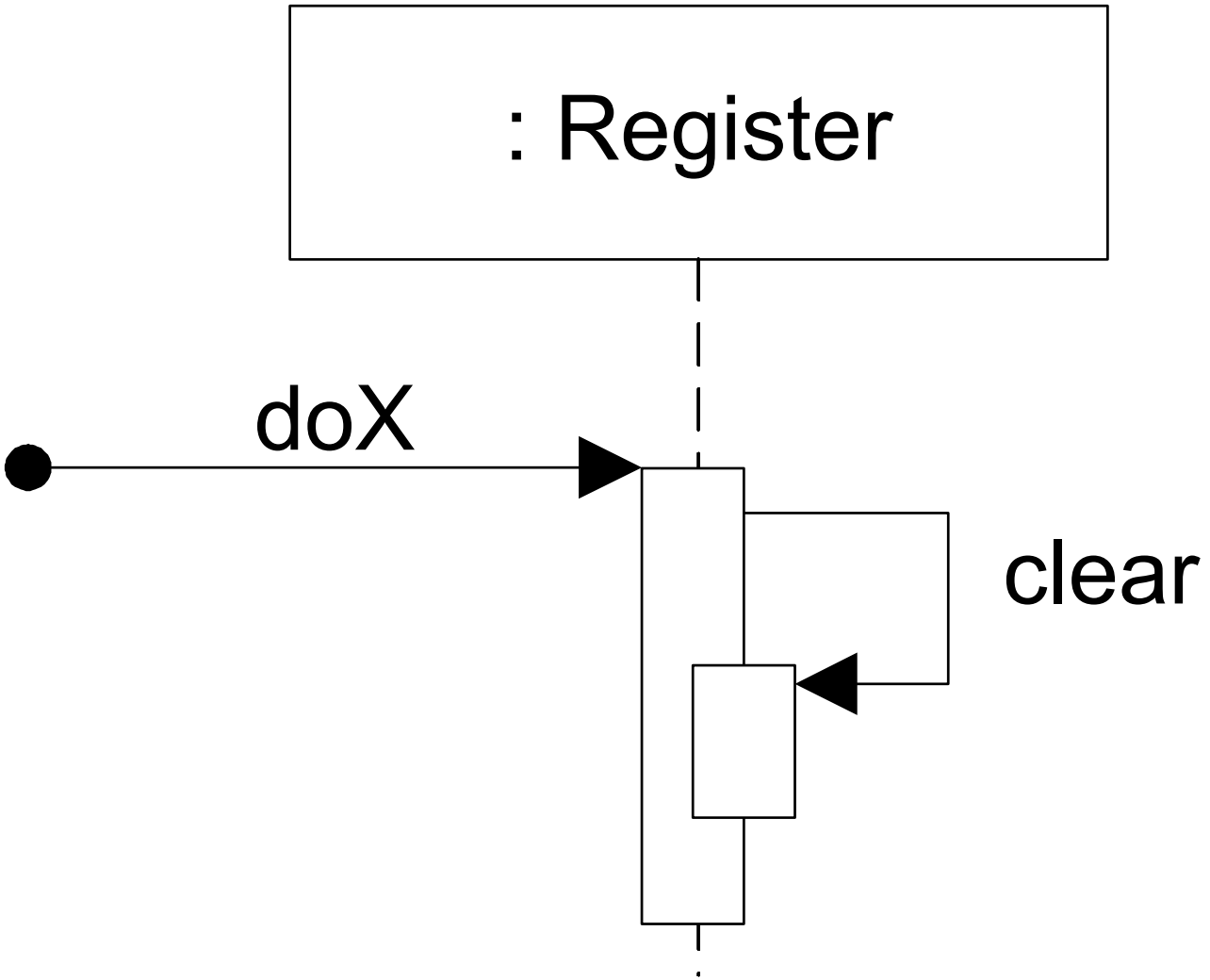
Proxy (Adapter)

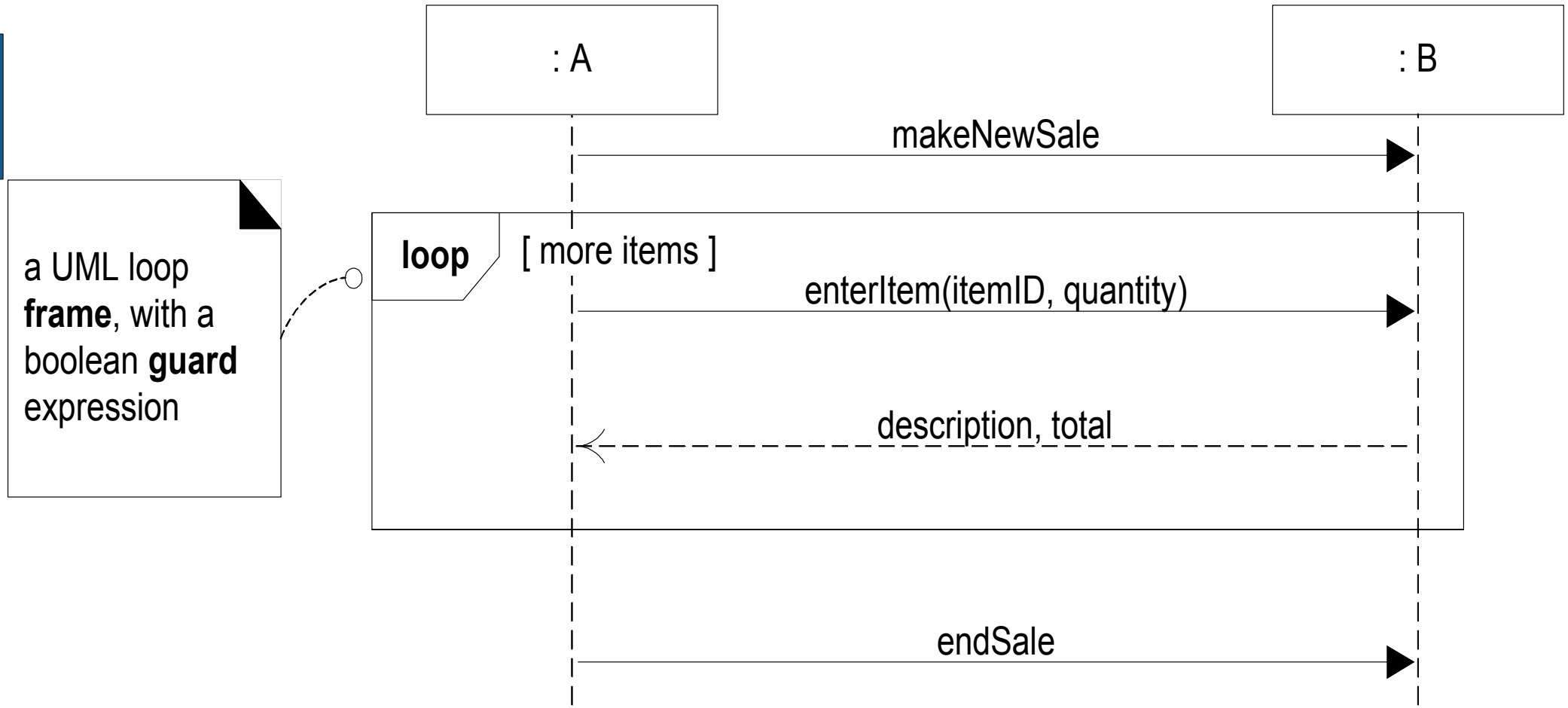


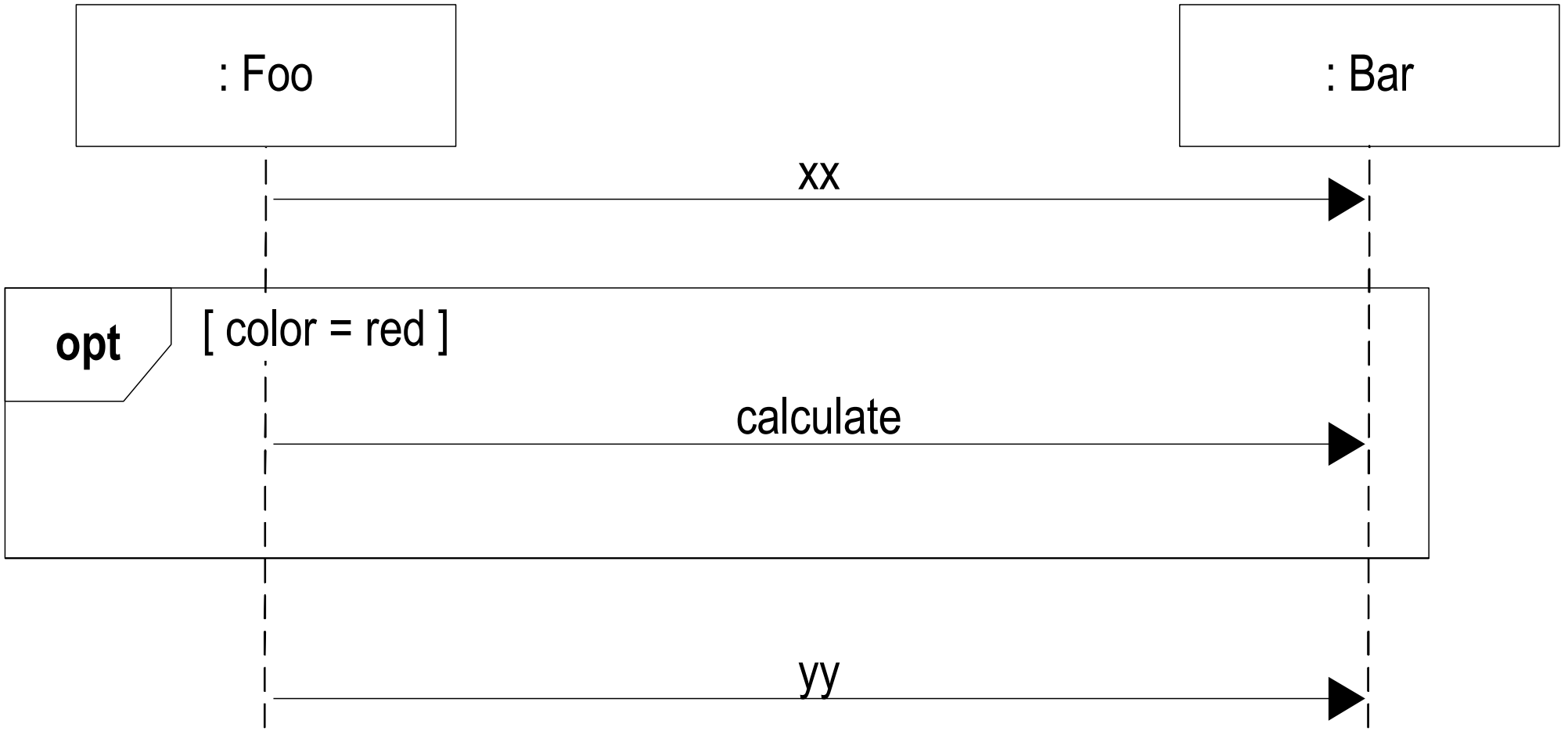

"accounting" actually references an instance of Accounting-RedirectionProxy
{
... payment work
if (payment completed)
completeSaleHandling()
}
1
Register
accounting : IAccountingAdapter
+ makePayment()
- completeSaleHandling()
«interface»
IAccountingAdapter
postReceivable(ReceivableEntry)
postSale(Sale)
...
1
2
{
...
accounting.postSale(currentSale)
...
}
2
AccountingRedirectionProxy
externalAccounting : IAccountingAdapter
localAccounting : IAccountingAdapter
postReceivable(ReceivableEntry)
postSale(Sale)
...
SAPAccountingAdapter
postReceivable(ReceivableEntry)
postSale(Sale)
...
{
externalAccounting.postSale(sale)
if (externalAccounting fails)
localAccounting.postSale(sale)
}
3
LocalAccounting
postReceivable(ReceivableEntry)
postSale(Sale)
...
{
save the sale in a local file (to be forwarded to external accounting later)
}
4


this is the Abstract Factory--an interface for creating a family of related objects
«interface»
IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
IBMJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
NCRJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
{
return new com.ibm.pos.jpos.CashDrawer()
}
{
return new com.ncr.posdevices.CashDrawer()
}
«interface»
jpos.CashDrawer
isDrawerOpened()
...
com.ibm.pos.jpos.CashDrawer
isDrawerOpened()
...
com.ncr.posdevices.CashDrawer
isDrawerOpened()
...

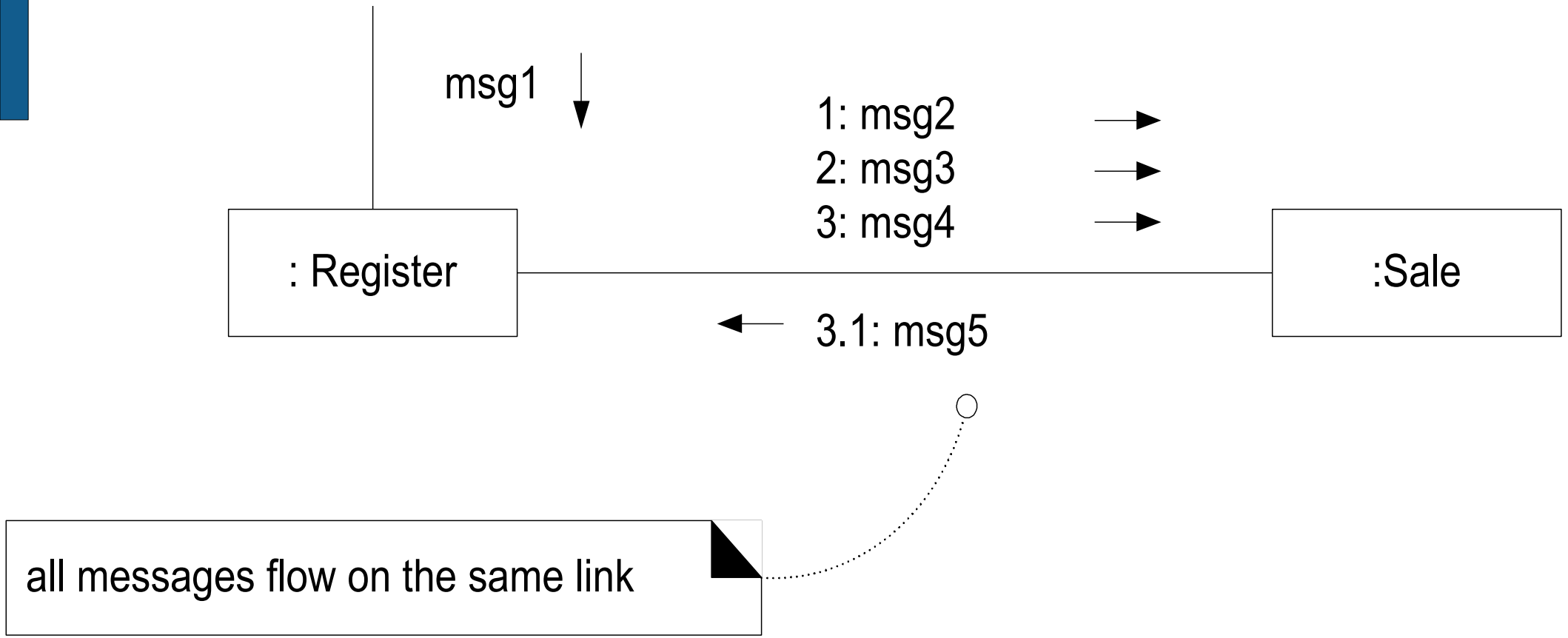

: Register
: Sale
doX
doA
doB
doC
doD
a **found message** whose sender will not be specified
execution specification bar indicates focus of control
typical **sychronous** message shown with a filled-arrow line


: Register
doX
clear

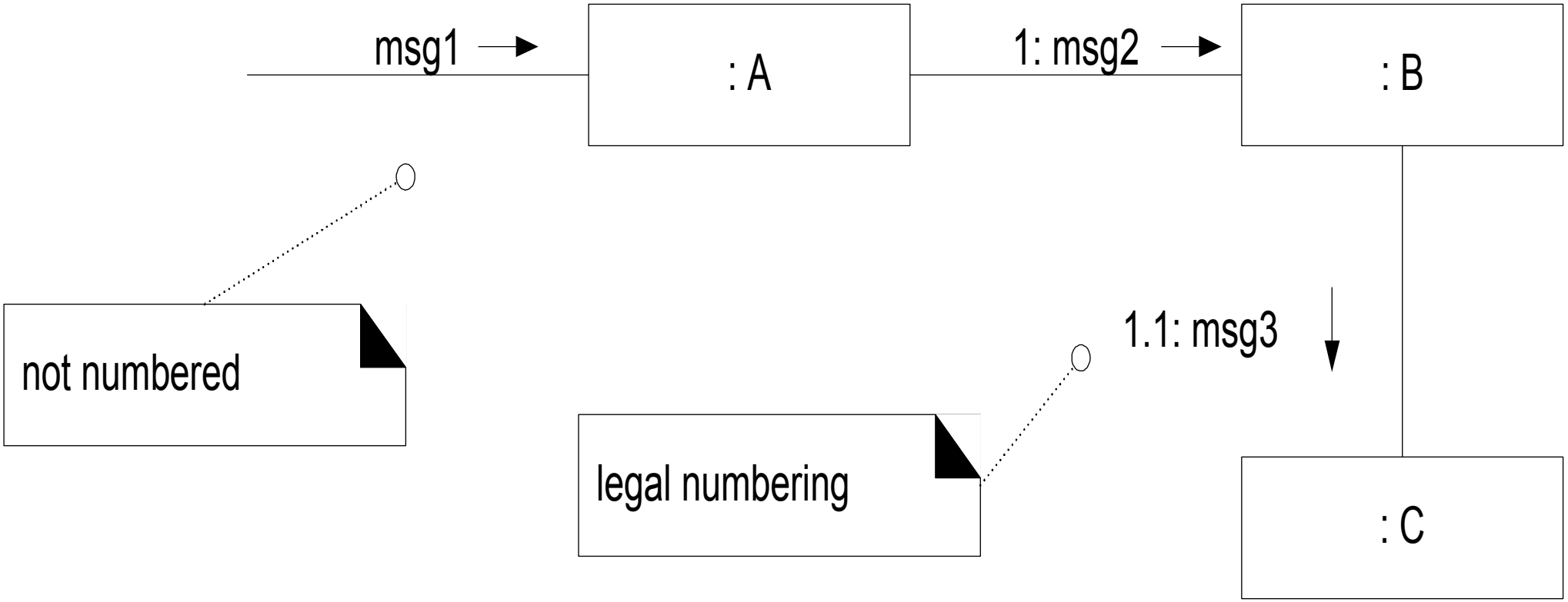

: A
: B
makeNewSale
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale
a UML loop **frame**, with a boolean **guard** expression


: Foo
: Bar
xx
opt
[color = red]
calculate
yy

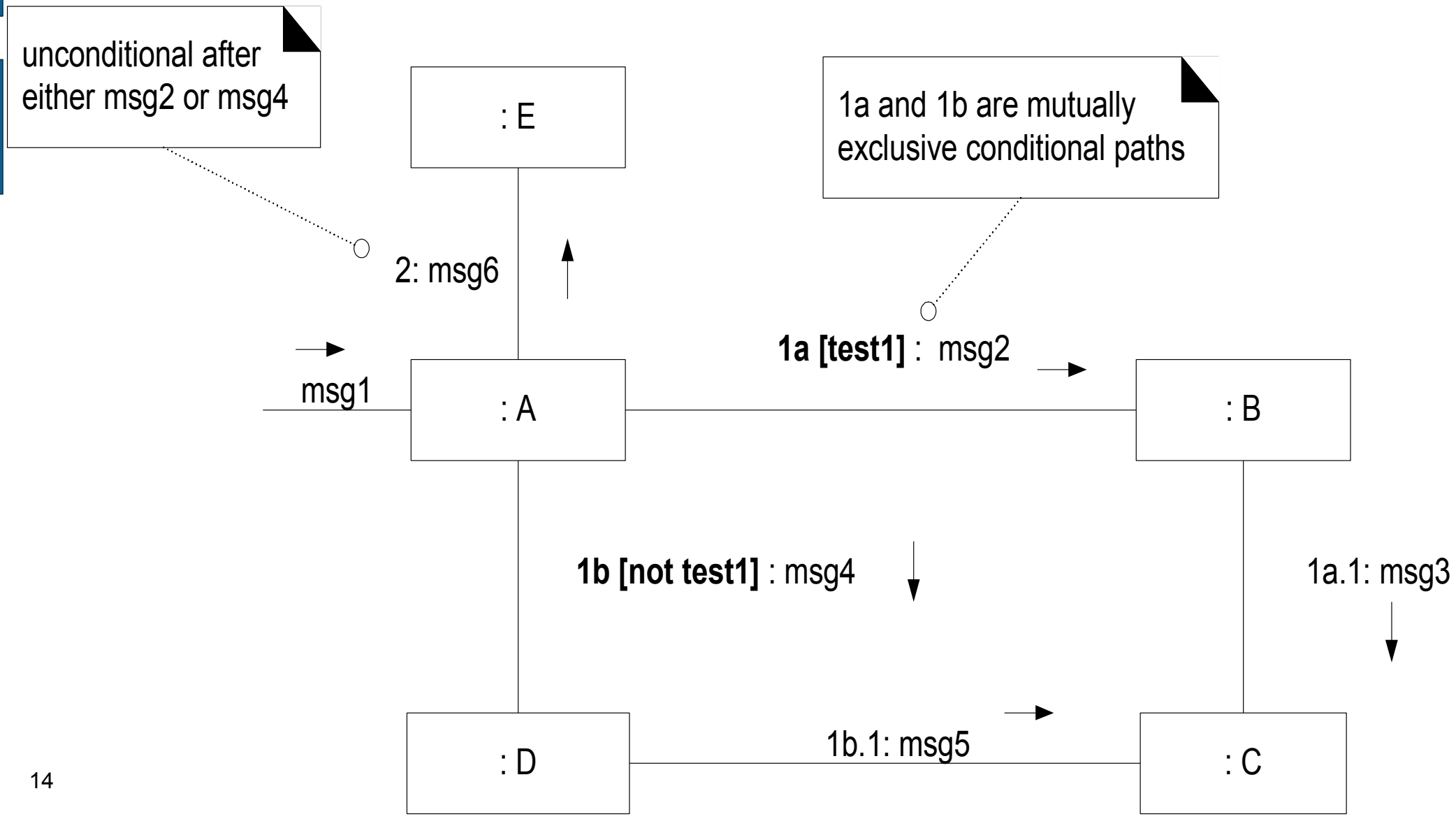
Nachrichten



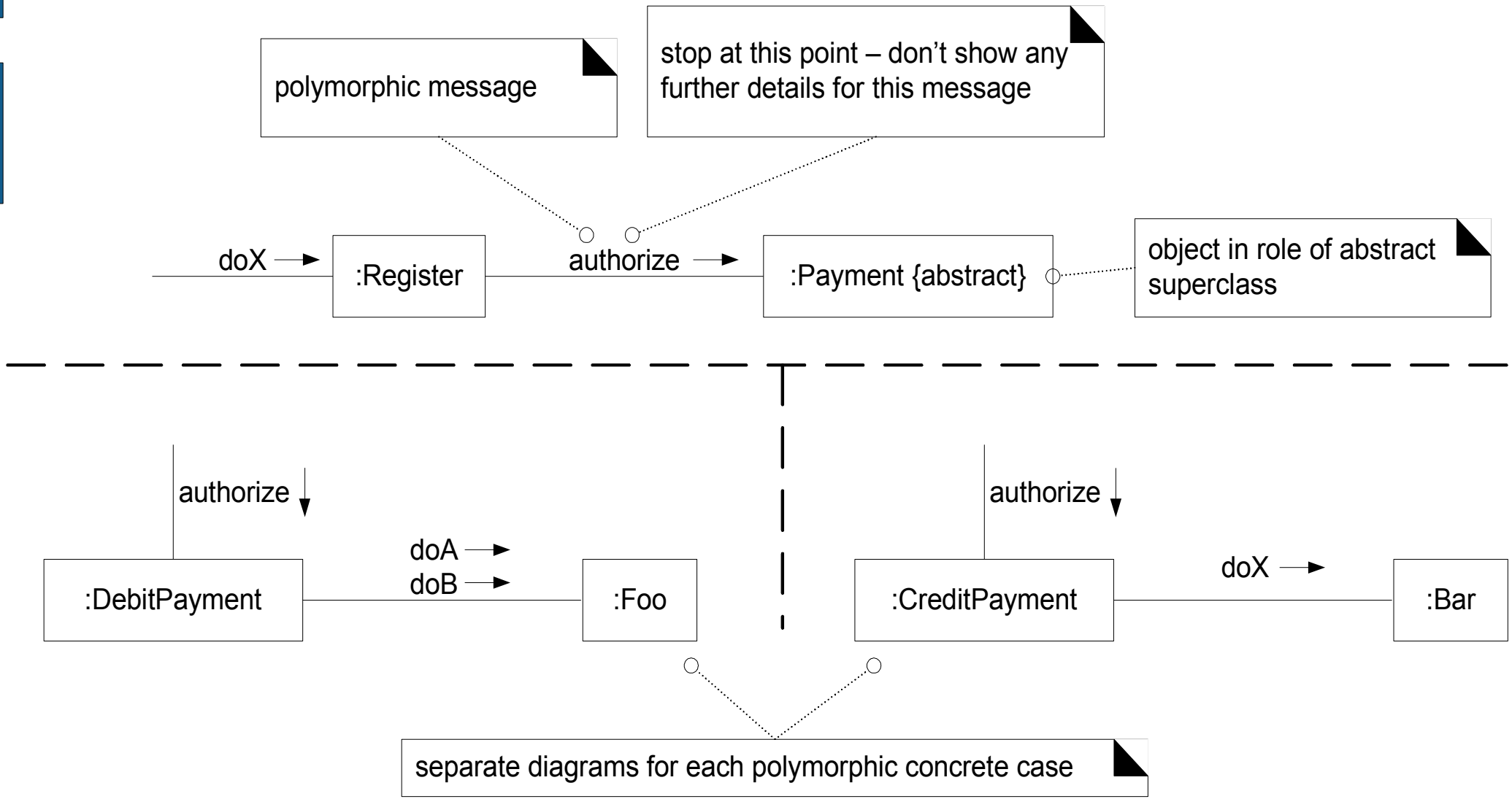
Verschachtelte Nummerierung

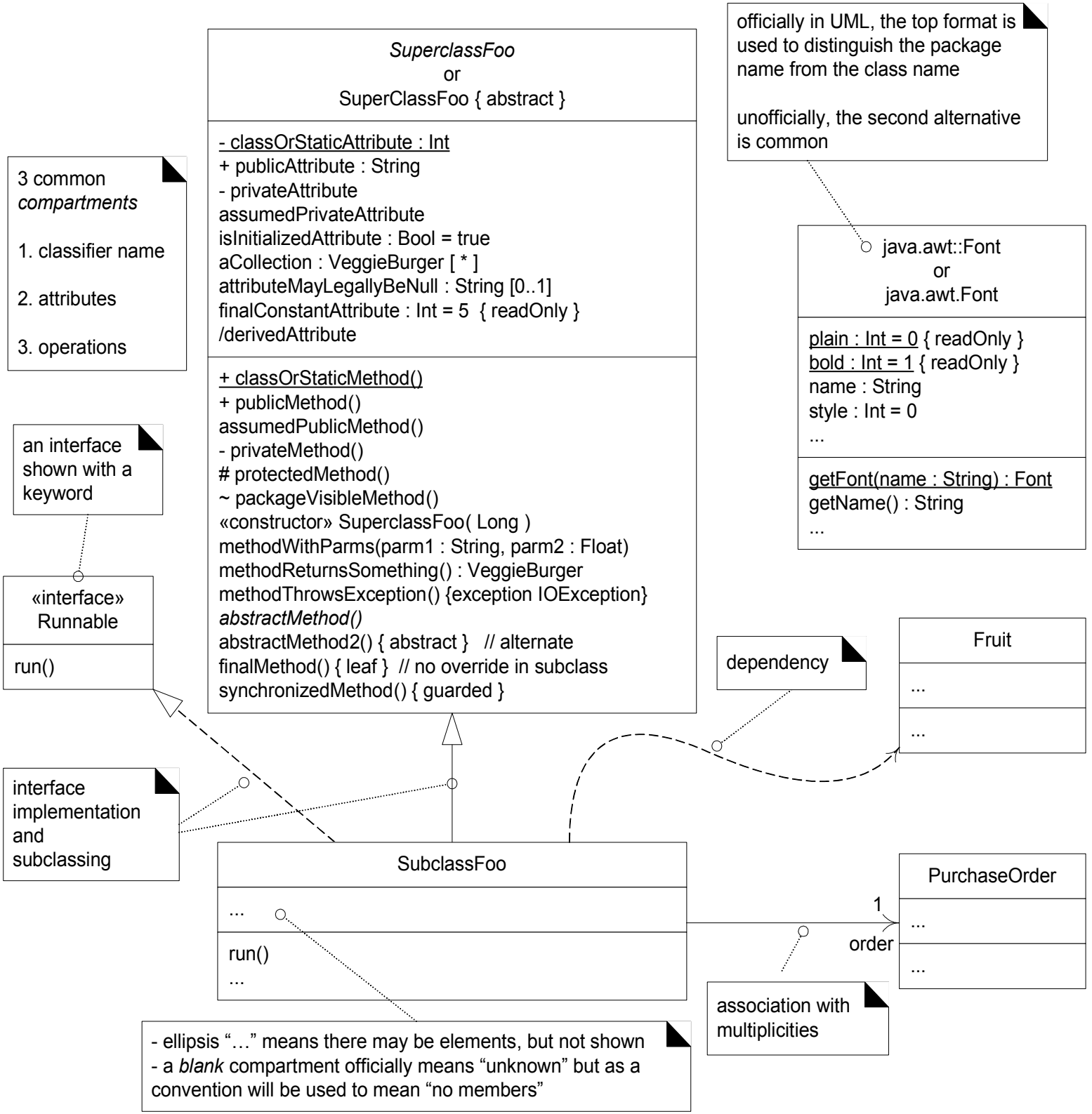


Bedingungen

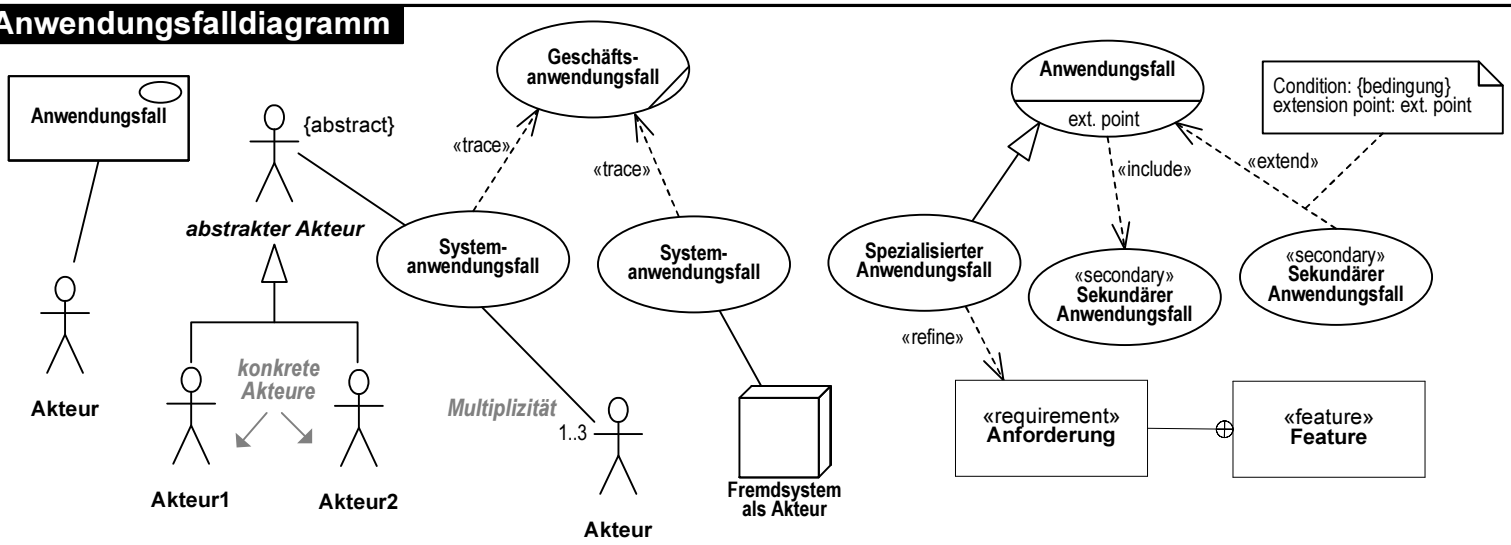


Polymorphismus

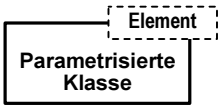



officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
3 common compartments
1. classifier name
2. attributes
3. operations
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
an interface shown with a keyword
«interface»
Runnable
run()
dependency
Fruit
...
...
interface implementation and subclassing
SubclassFoo
...
run()
...
PurchaseOrder
...
...
1
order
association with multiplicities
- ellipsis "…" means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

Anwendungsfalldiagramm



Klassendiagramm

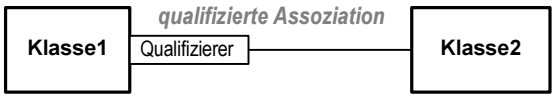


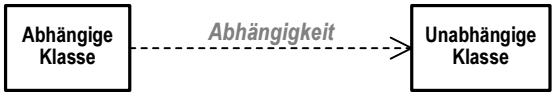
Syntax für Attribute:
Sichtbarkeit Attributname : Paket::Typ [Multiplizität Ordnung] = Initialwert {Eigenschaftswerte}
Eigenschaftswerte: {readOnly}, {ordered}, {composite}

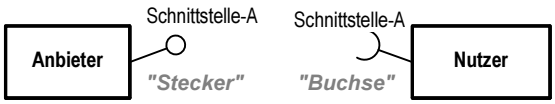
Syntax für Operationen:
Sichtbarkeit Operationsname (Parameterliste):Rückgabetyp {Eigenschaftswerte}

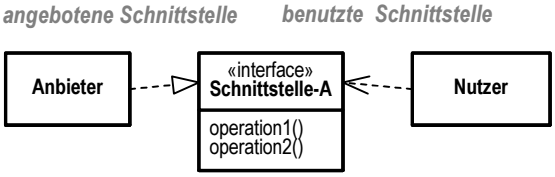
Sichtbarkeit:
+ public element
# protected element
– private element
~ package element

Parameterliste: Richtung Name : Typ = Standardwert
Eigenschaftswerte: {query}
Richtung: in, out, inout

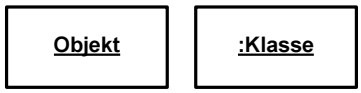


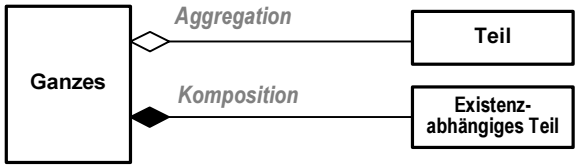


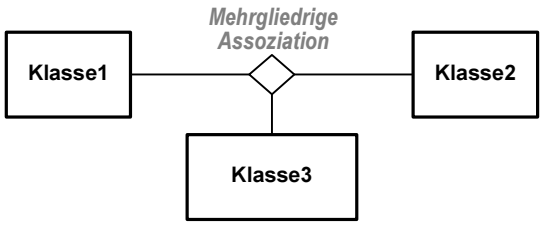


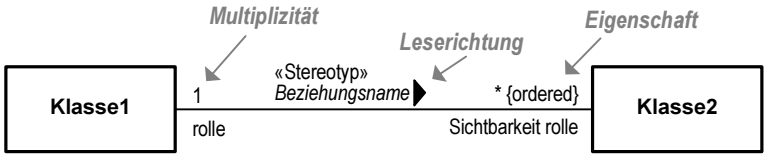


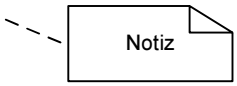
Objektdiagramm



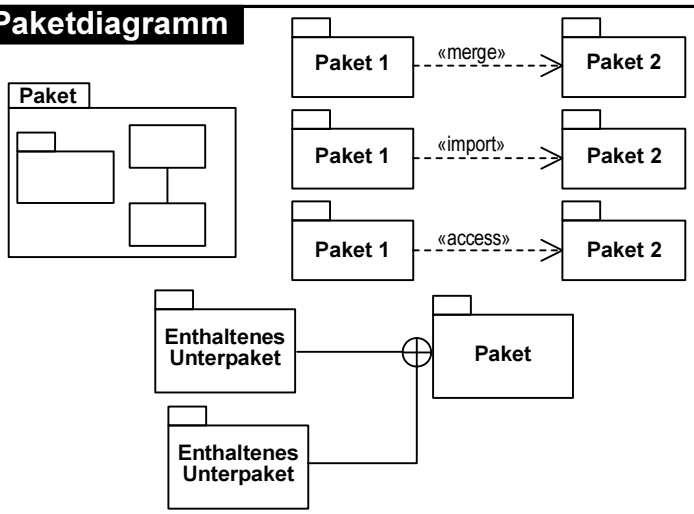




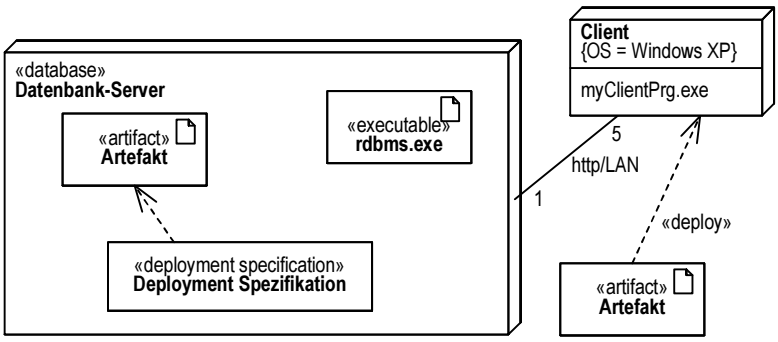




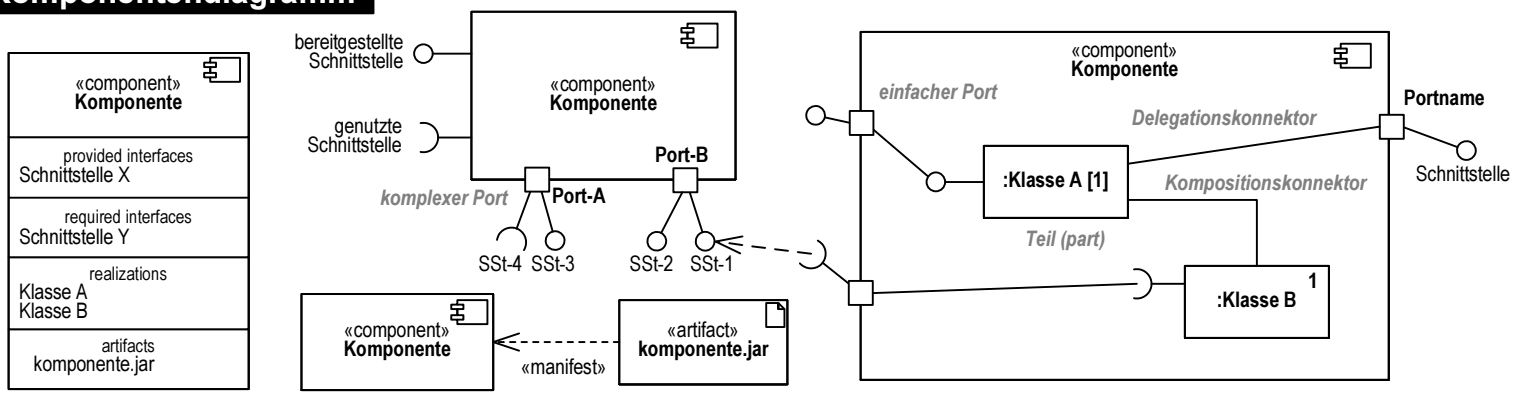
Paketdiagramm



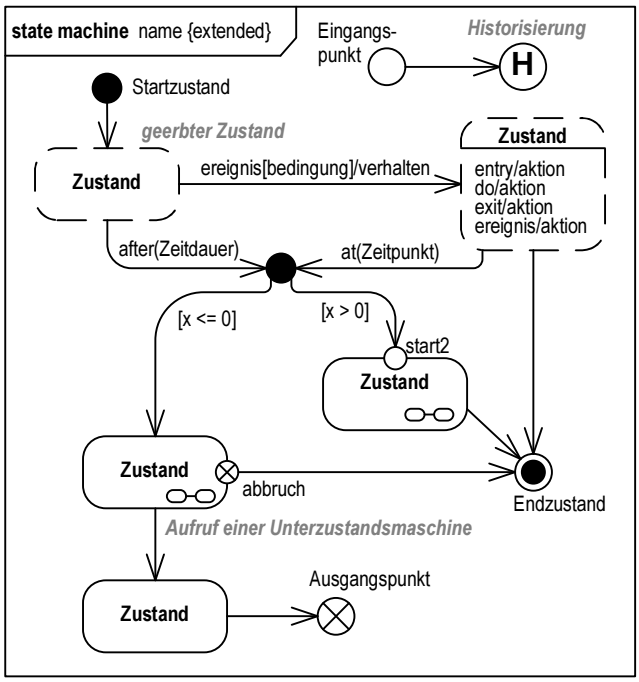
Einsatz- und Verteilungsdiagramm

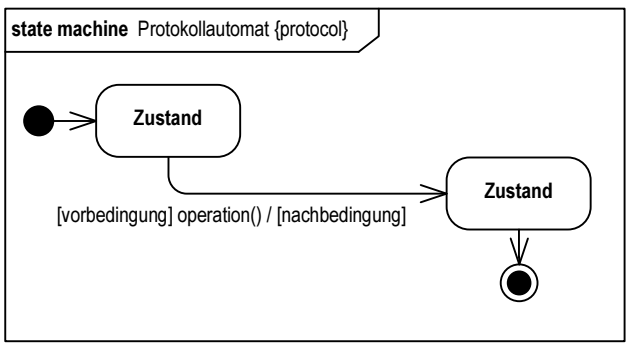


Komponentendiagramm

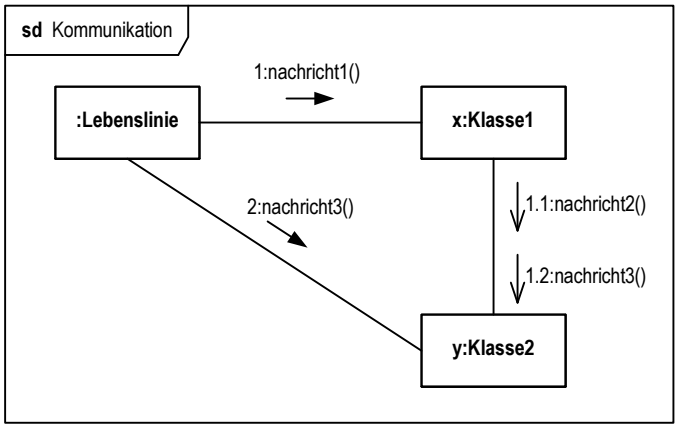


Zustandsdiagramm

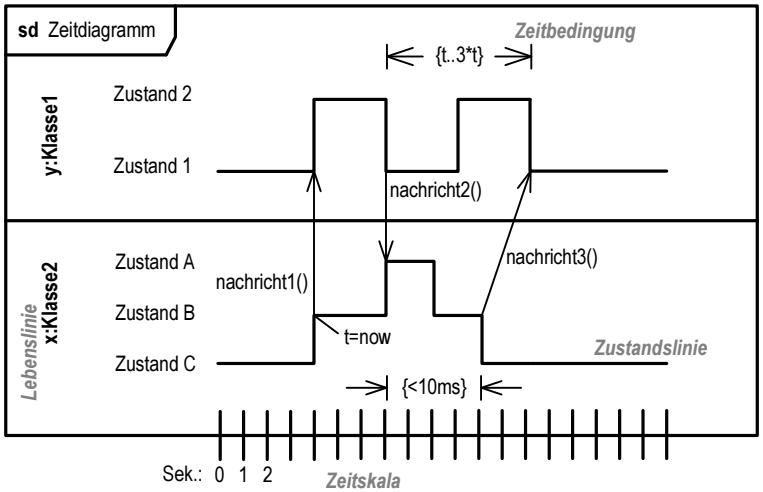




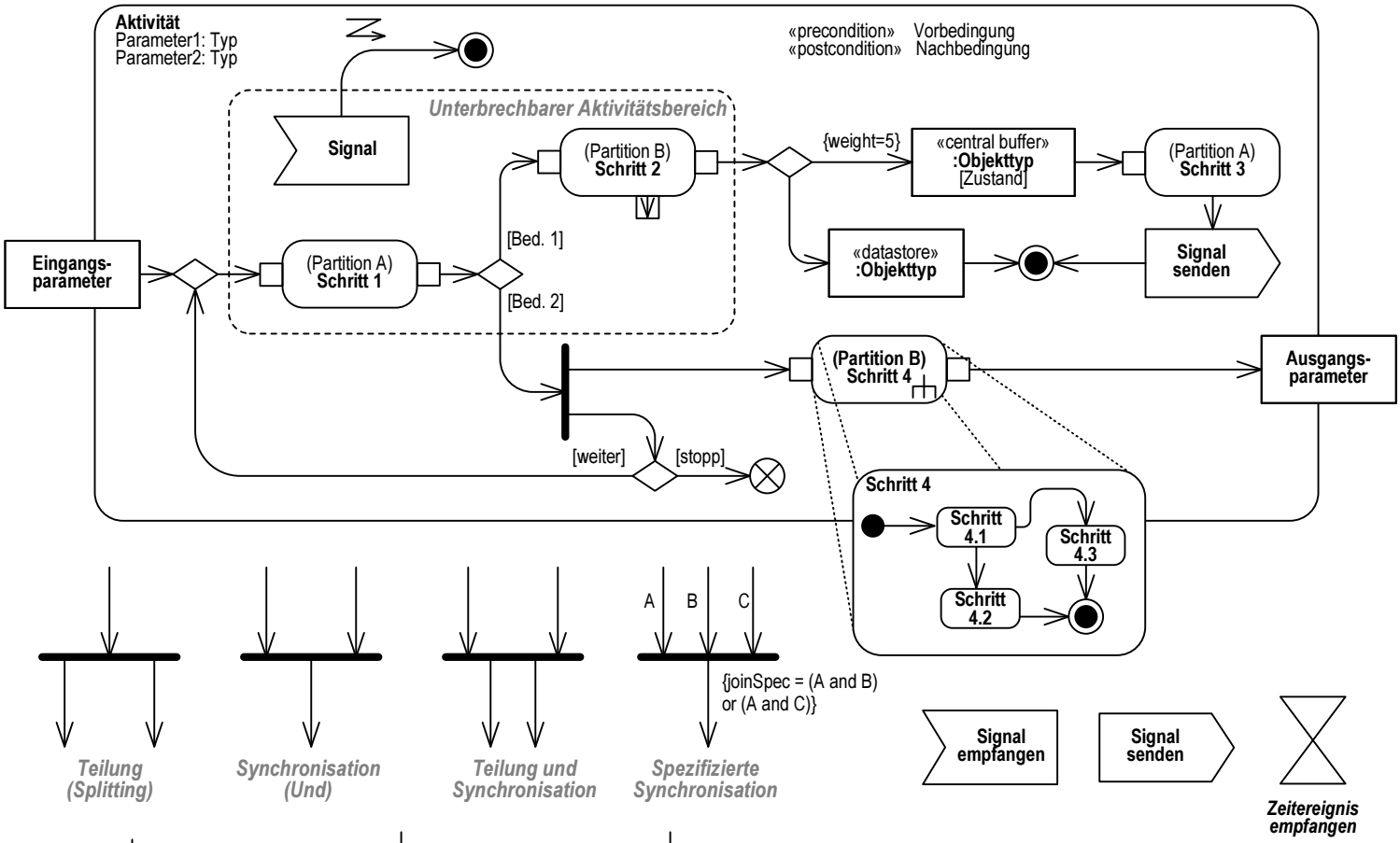
Kommunikationsdiagramm



Zeitdiagramm

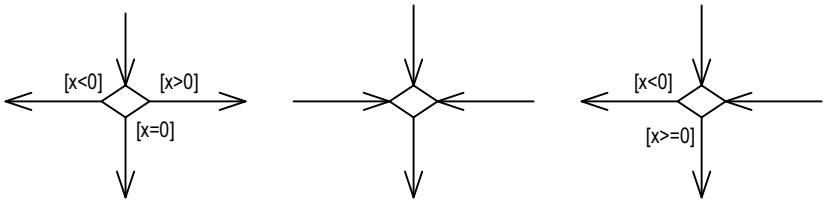


Aktivitätsdiagramm

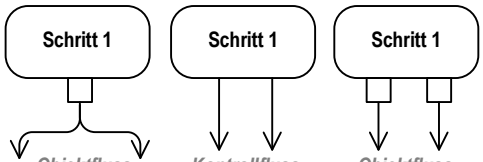


Teilung (Splitting) | *Synchronisation (Und)* | *Teilung und Synchronisation* | *Spezifizierte Synchronisation*

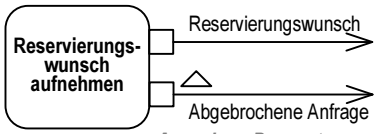
Zeitereignis empfangen



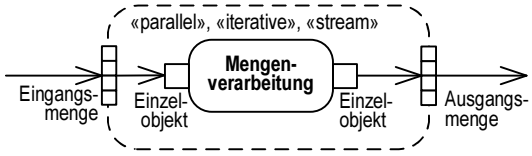
Entscheidung | *Zusammenführung (Oder)* | *Entscheidung und Zusammenführung*

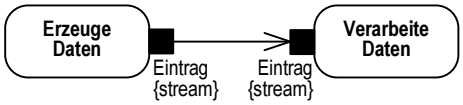


Objektfluss Oder-Semantik | *Kontrollfluss Und-Semantik* | *Objektfluss Und-Semantik*

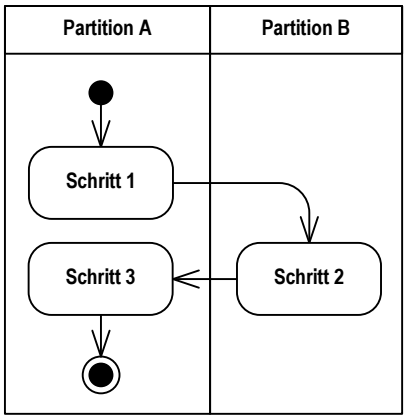


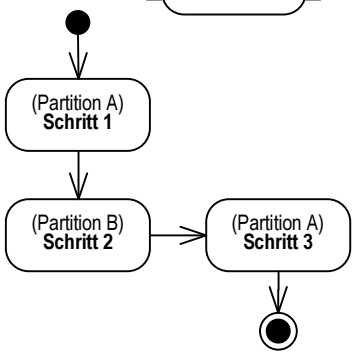
Ausnahme-Parameter (Exception)



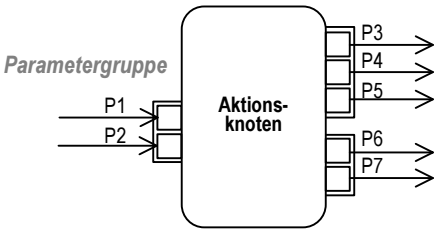


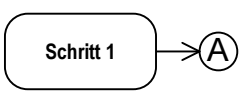
Kurzschreibweise



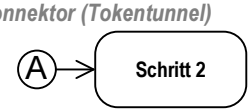


Startknoten | *Endknoten* | *Ablaufende*

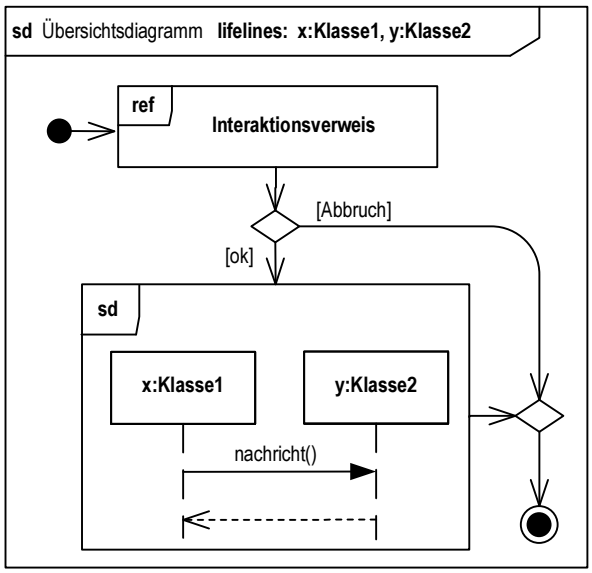




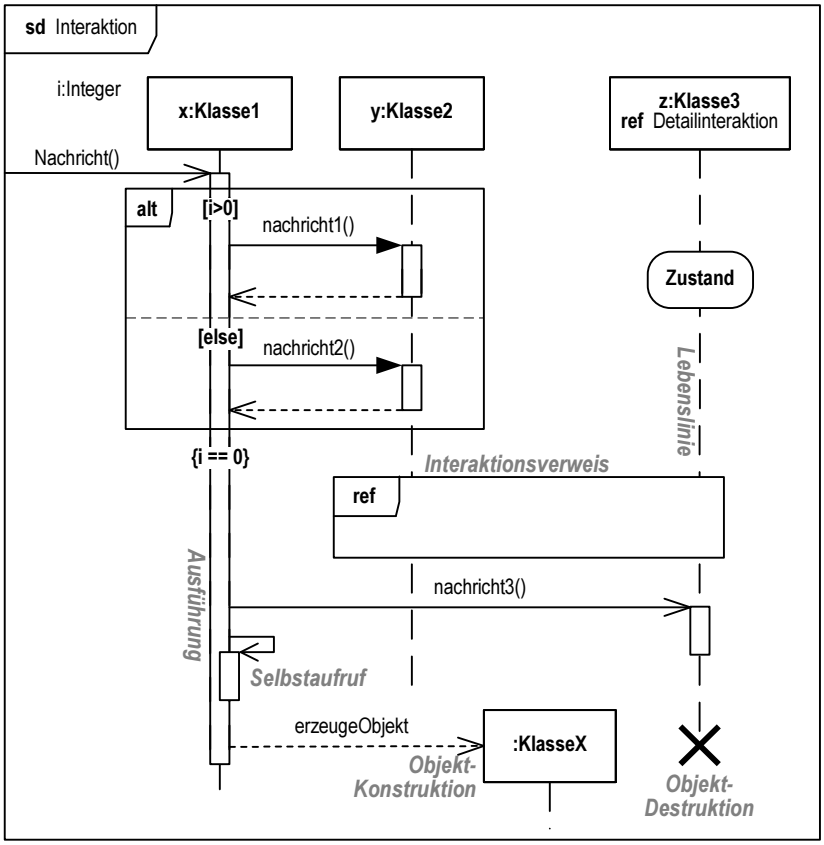
Konnektor (Tokentunnel)

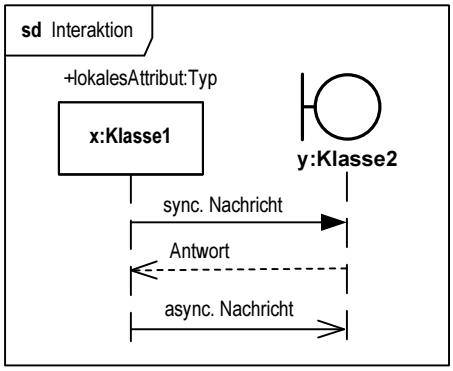


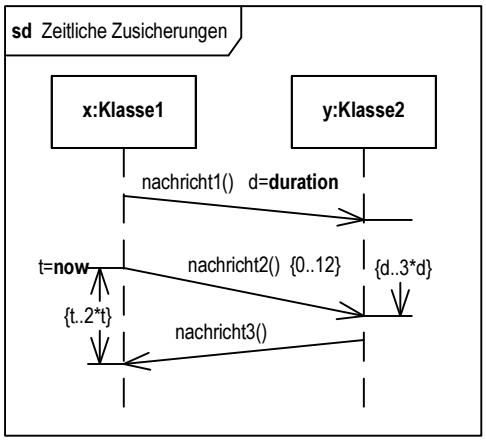
Interaktionsübersicht



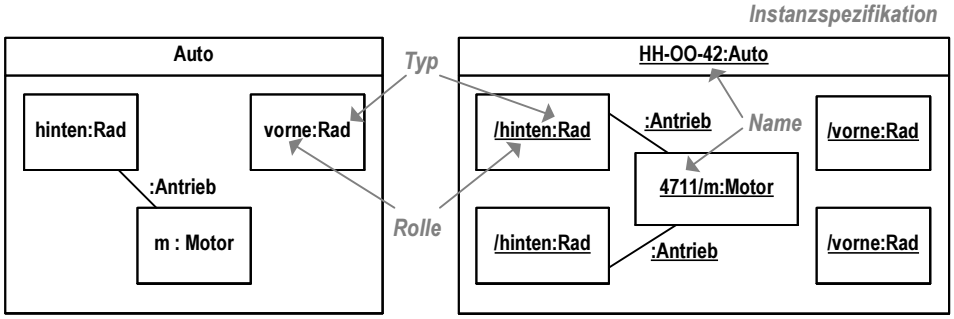
Sequenzdiagramm

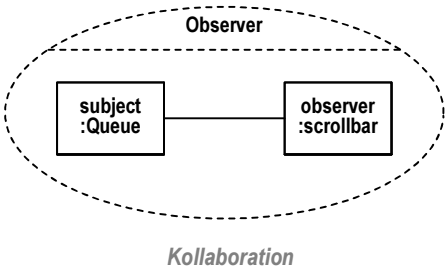




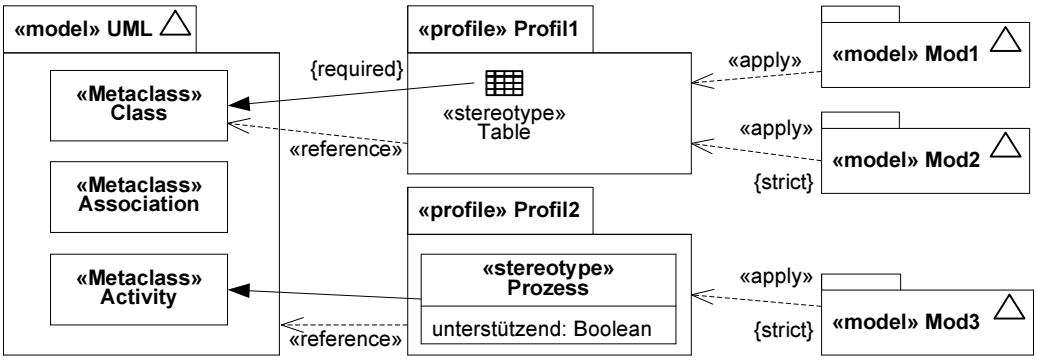


Kompositionsstrukturdiagramm



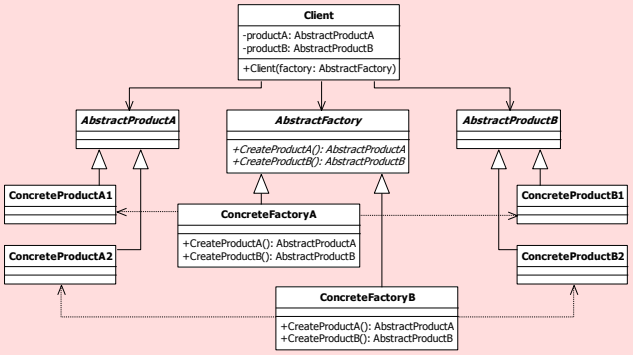


Profildiagramm



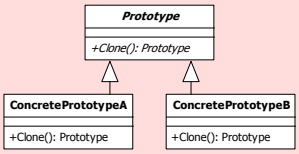
Creational Design Patterns

Abstract Factory



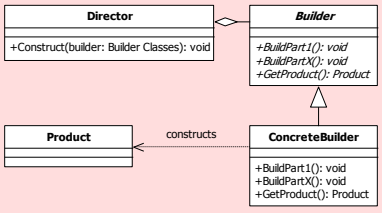
The abstract factory pattern is used to provide a client with a set of related or dependant objects. The "family" of objects created by the factory are determined at run-time.

Prototype



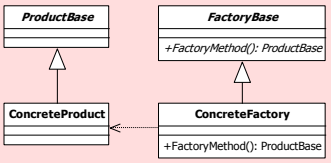
The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone. This practise is particularly useful when the construction of a new object is inefficient.

Builder



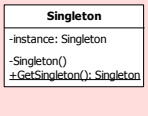
The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.

Factory Method



The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.

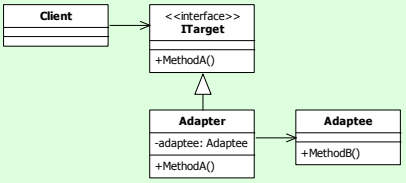
Singleton



The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance.

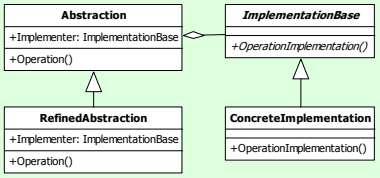
Structural Design Patterns

Adapter



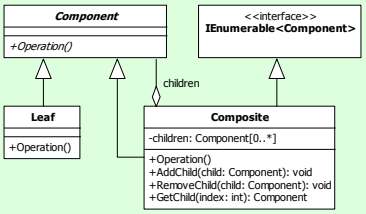
The adapter pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

Bridge



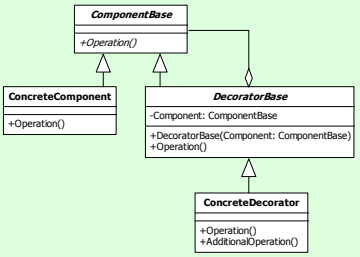
The bridge pattern is used to separate the abstract elements of a class from the implementation details, providing the means to replace the implementation details without modifying the abstraction.

Composite



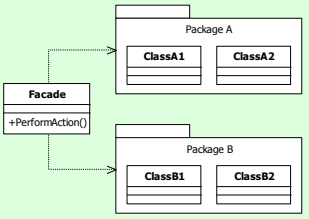
The composite pattern is used to create hierarchical, recursive tree structures of related objects where any element of the structure may be accessed and utilised in a standard manner.

Decorator



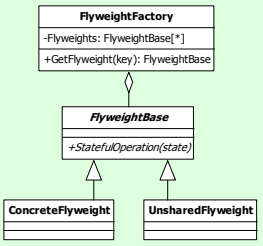
The decorator pattern is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behaviour.

Facade



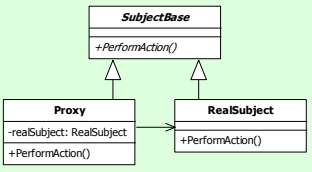
The facade pattern is used to define a simplified interface to a more complex subsystem.

Flyweight



The flyweight pattern is used to reduce the memory and resource usage for complex models containing many hundreds, thousands or hundreds of thousands of similar objects.

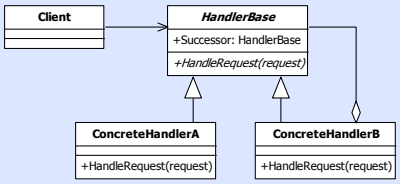
Proxy



The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. The proxy provides the same public interface as the underlying subject class, adding a level of indirection by accepting requests from a client object and passing these to the real subject object as necessary.

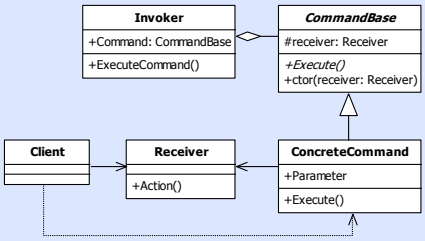
Behavioural Design Patterns

Chain of Responsibility



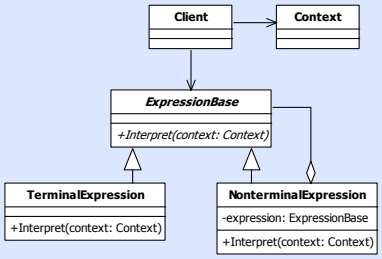
The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler.

Command



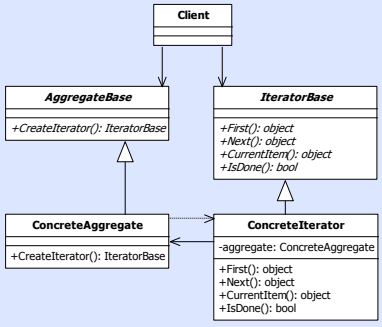
The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.

Interpreter



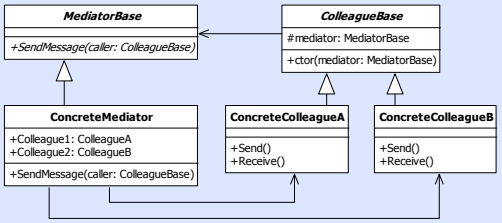
The interpreter pattern is used to define the grammar for instructions that form part of a language or notation, whilst allowing the grammar to be easily extended.

Iterator



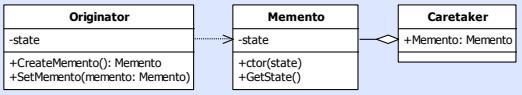
The iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure.

Mediator



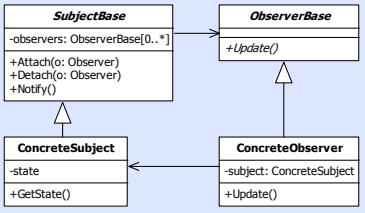
The mediator pattern is used to reduce coupling between classes that communicate with each other. Instead of classes communicating directly, and thus requiring knowledge of their implementation, the classes send messages via a mediator object.

Memento



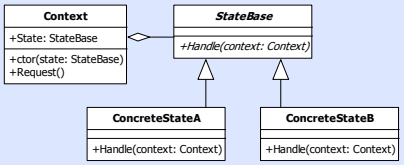
The memento pattern is used to capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.

Observer



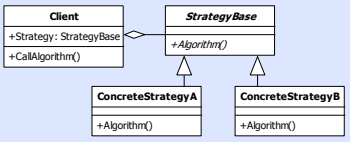
The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.

State



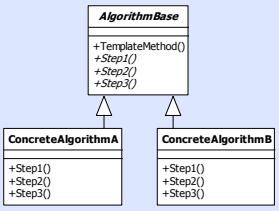
The state pattern is used to alter the behaviour of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.

Strategy



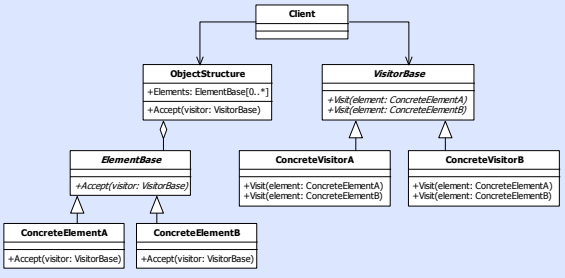
The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.

Template Method



The template method pattern is used to define the basic steps of an algorithm and allow the implementation of the individual steps to be changed.

Visitor



The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

DZone Refcardz

CONTENTS INCLUDE:

Design Patterns

By Jason McDonald

ABOUT DESIGN PATTERNS

This Design Patterns refcard provides a quick reference to the original 23 Gang of Four design patterns, as listed in the book *Design Patterns: Elements of Reusable Object-Oriented Software*. Each pattern includes class diagrams, explanation, usage information, and a real world example.

Creational Patterns: Used to construct objects such that they can be decoupled from their implementing system.

Structural Patterns: Used to form large object structures between many disparate objects.

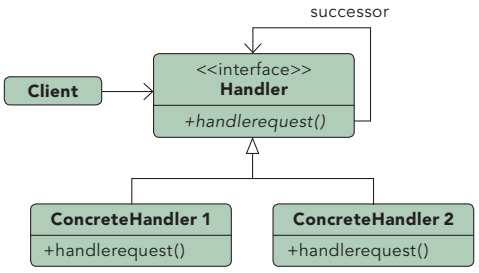
Behavioral Patterns: Used to manage algorithms, relationships, and responsibilities between objects.

Object Scope: Deals with object relationships that can be changed at runtime.

Class Scope: Deals with class relationships that can be changed at compile time.

C Abstract Factory	S Decorator	C Prototype
S Adapter	S Facade	S Proxy
S Bridge	C Factory Method	B Observer
C Builder	S Flyweight	C Singleton
B Chain of Responsibility	B Interpreter	B State
B Command	B Iterator	B Strategy
S Composite	B Mediator	B Template Method
	B Memento	B Visitor

CHAIN OF RESPONSIBILITY — Object Behavioral



Purpose

Gives more than one object an opportunity to handle a request by linking receiving objects together.

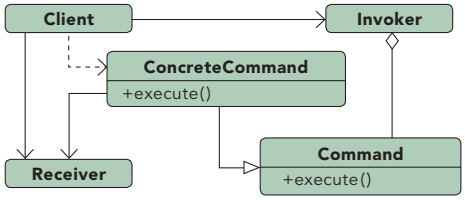
Use When

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

Example

Exception handling in some languages implements this pattern. When an exception is thrown in a method the runtime checks to see if the method has a mechanism to handle the exception or if it should be passed up the call stack. When passed up the call stack the process repeats until code to handle the exception is encountered or until there are no more parent objects to hand the request to.

COMMAND — Object Behavioral



Purpose

Encapsulates a request allowing it to be treated as an object. This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

Use When

- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

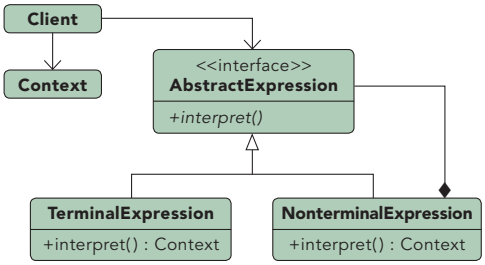
Example

Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is enqueued implements its particular algorithm within the confines of the interface the queue is expecting.

Design Patterns

www.dzone.com

INTERPRETER — Class Behavioral



Purpose

Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar.

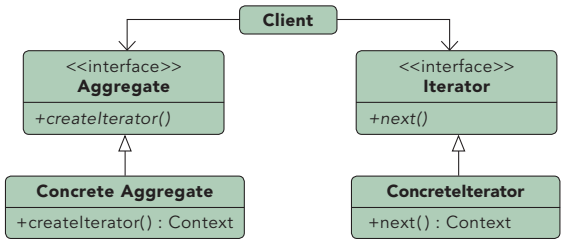
Use When

- There is grammar to interpret that can be represented as large syntax trees.
- The grammar is simple.
- Efficiency is not important.
- Decoupling grammar from underlying expressions is desired.

Example

Text based adventures, wildly popular in the 1980's, provide a good example of this. Many had simple commands, such as "step down" that allowed traversal of the game. These commands could be nested such that it altered their meaning. For example, "go in" would result in a different outcome than "go up". By creating a hierarchy of commands based upon the command and the qualifier (non-terminal and terminal expressions) the application could easily map many command variations to a relating tree of actions.

ITERATOR — Object Behavioral



Purpose

Allows for access to the elements of an aggregate object without allowing access to its underlying representation.

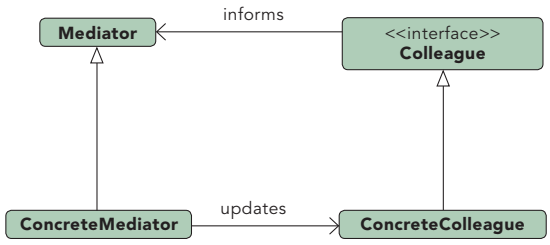
Use When

- Access to elements is needed without access to the entire representation.
- Multiple or concurrent traversals of the elements are needed.
- A uniform interface for traversal is needed.
- Subtle differences exist between the implementation details of various iterators.

Example

The Java implementation of the iterator pattern allows users to traverse various types of data sets without worrying about the underlying implementation of the collection. Since clients simply interact with the iterator interface, collections are left to define the appropriate iterator for themselves. Some will allow full access to the underlying data set while others may restrict certain functionalities, such as removing items.

MEDIATOR — Object Behavioral



Purpose

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

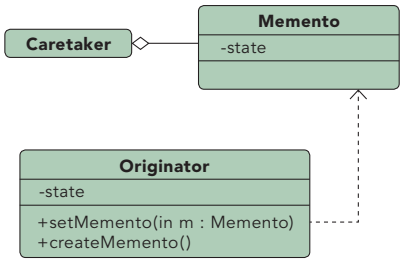
Use When

- Communication between sets of objects is well defined and complex.
- Too many relationships exist and common point of control or communication is needed.

Example

Mailing list software keeps track of who is signed up to the mailing list and provides a single point of access through which any one person can communicate with the entire list. Without a mediator implementation a person wanting to send a message to the group would have to constantly keep track of who was signed up and who was not. By implementing the mediator pattern the system is able to receive messages from any point then determine which recipients to forward the message on to, without the sender of the message having to be concerned with the actual recipient list.

MEMENTO — Object Behavioral



Purpose

Allows for capturing and externalizing an object's internal state so that it can be restored later, all without violating encapsulation.

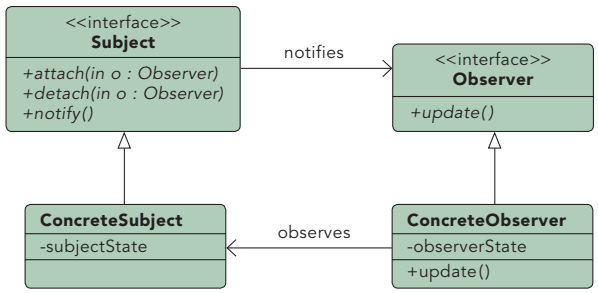
Use When

- The internal state of an object must be saved and restored at a later time.
- Internal state cannot be exposed by interfaces without exposing implementation.
- Encapsulation boundaries must be preserved.

Example

Undo functionality can nicely be implemented using the memento pattern. By serializing and deserializing the state of an object before the change occurs we can preserve a snapshot of it that can later be restored should the user choose to undo the operation.

OBSERVER — Object Behavioral



Purpose

Lets one or more objects be notified of state changes in other objects within the system.

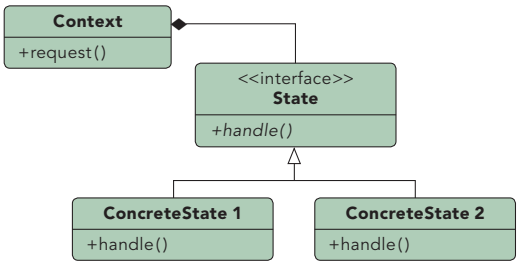
Use When

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.

Example

This pattern can be found in almost every GUI environment. When buttons, text, and other fields are placed in applications the application typically registers as a listener for those controls. When a user triggers an event, such as clicking a button, the control iterates through its registered observers and sends a notification to each.

STATE — Object Behavioral



Purpose

Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.

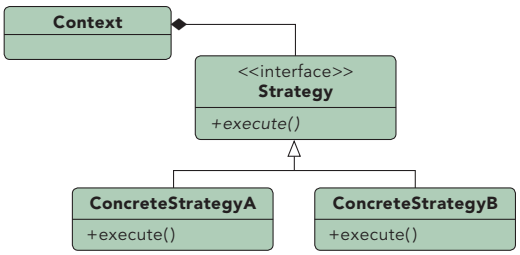
Use When

- The behavior of an object should be influenced by its state.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example

An email object can have various states, all of which will change how the object handles different functions. If the state is "not sent" then the call to send() is going to send the message while a call to recallMessage() will either throw an error or do nothing. However, if the state is "sent" then the call to send() would either throw an error or do nothing while the call to recallMessage() would attempt to send a recall notification to recipients. To avoid conditional statements in most or all methods there would be multiple state objects that handle the implementation with respect to their particular state. The calls within the Email object would then be delegated down to the appropriate state object for handling.

STRATEGY — Object Behavioral



Purpose

Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

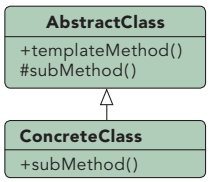
Use When

- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

Example

When importing data into a new system different validation algorithms may be run based on the data set. By configuring the import to utilize strategies the conditional logic to determine what validation set to run can be removed and the import can be decoupled from the actual validation code. This will allow us to dynamically call one or more strategies during the import.

TEMPLATE METHOD — Class Behavioral



Purpose

Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.

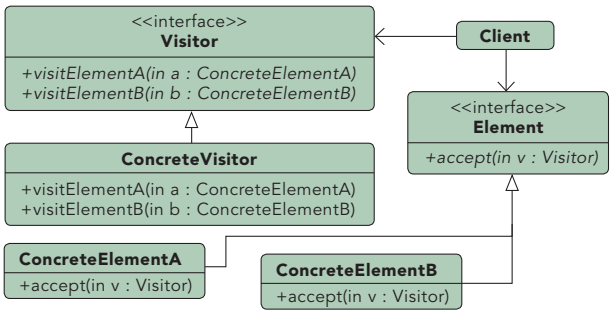
Use When

- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most or all subclasses need to implement the behavior.

Example

A parent class, InstantMessage, will likely have all the methods required to handle sending a message. However, the actual serialization of the data to send may vary depending on the implementation. A video message and a plain text message will require different algorithms in order to serialize the data correctly. Subclasses of InstantMessage can provide their own implementation of the serialization method, allowing the parent class to work with them without understanding their implementation details.

VISITOR — Object Behavioral



Purpose

Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.

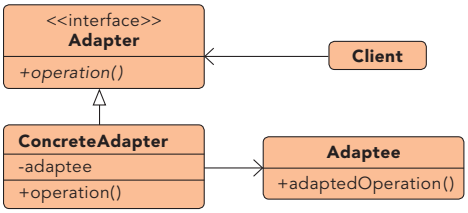
Use When

- An object structure must have many unrelated operations performed upon it.
- The object structure can't change but operations performed on it can.
- Operations must be performed on the concrete classes of an object structure.
- Exposing internal state or operations of the object structure is acceptable.
- Operations should be able to operate on multiple object structures that implement the same interface sets.

Example

Calculating taxes in different regions on sets of invoices would require many different variations of calculation logic. Implementing a visitor allows the logic to be decoupled from the invoices and line items. This allows the hierarchy of items to be visited by calculation code that can then apply the proper rates for the region. Changing regions is as simple as substituting a different visitor.

ADAPTER — Class and Object Structural



Purpose

Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.

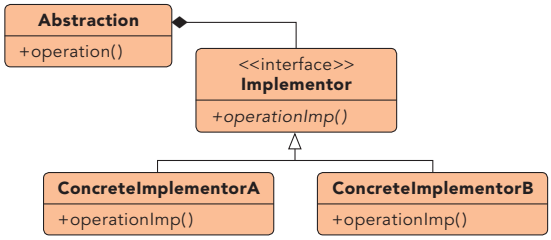
Use When

- A class to be used doesn't meet interface requirements.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example

A billing application needs to interface with an HR application in order to exchange employee data, however each has its own interface and implementation for the Employee object. In addition, the SSN is stored in different formats by each system. By creating an adapter we can create a common interface between the two applications that allows them to communicate using their native objects and is able to transform the SSN format in the process.

BRIDGE — Object Structural



Purpose

Defines an abstract object structure independently of the implementation object structure in order to limit coupling.

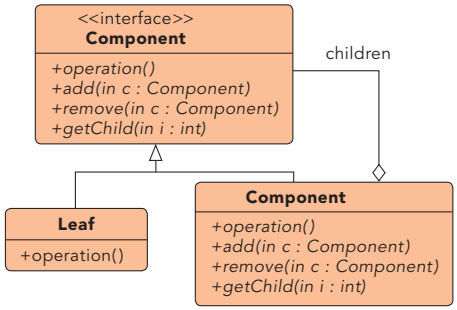
Use When

- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

Example

The Java Virtual Machine (JVM) has its own native set of functions that abstract the use of windowing, system logging, and byte code execution but the actual implementation of these functions is delegated to the operating system the JVM is running on. When an application instructs the JVM to render a window it delegates the rendering call to the concrete implementation of the JVM that knows how to communicate with the operating system in order to render the window.

COMPOSITE — Object Structural



Purpose

Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

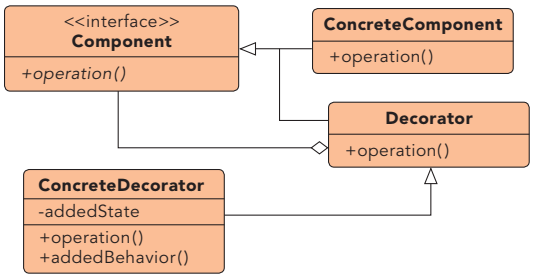
Use When

- Hierarchical representations of objects are needed..
- Objects and compositions of objects should be treated uniformly.

Example

Sometimes the information displayed in a shopping cart is the product of a single item while other times it is an aggregation of multiple items. By implementing items as composites we can treat the aggregates and the items in the same way, allowing us to simply iterate over the tree and invoke functionality on each item. By calling the getCost() method on any given node we would get the cost of that item plus the cost of all child items, allowing items to be uniformly treated whether they were single items or groups of items.

DECORATOR — Object Structural



Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

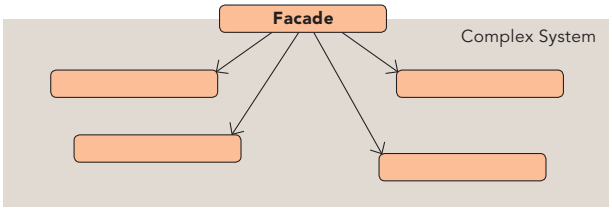
Use When

- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Subclassing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.

Example

Many businesses set up their mail systems to take advantage of decorators. When messages are sent from someone in the company to an external address the mail server decorates the original message with copyright and confidentiality information. As long as the message remains internal the information is not attached. This decoration allows the message itself to remain unchanged until a runtime decision is made to wrap the message with additional information.

FACADE — Object Structural



Purpose

Supplies a single interface to a set of interfaces within a system.

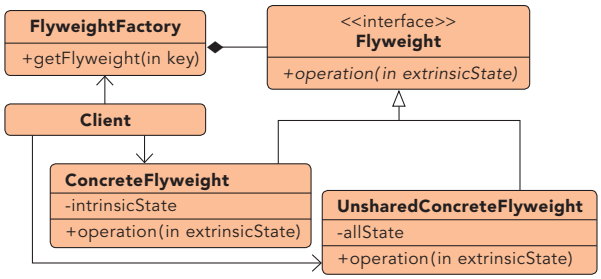
Use When

- A simple interface is needed to provide access to a complex system.
- There are many dependencies between system implementations and clients.
- Systems and subsystems should be layered.

Example

By exposing a set of functionalities through a web service the client code needs to only worry about the simple interface being exposed to them and not the complex relationships that may or may not exist behind the web service layer. A single web service call to update a system with new data may actually involve communication with a number of databases and systems, however this detail is hidden due to the implementation of the façade pattern.

FLYWEIGHT — Object Structural



Purpose

Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.

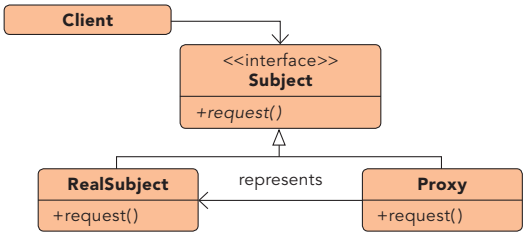
Use When

- Many like objects are used and storage cost is high.
- The majority of each object's state can be made extrinsic.
- A few shared objects can replace many unshared ones.
- The identity of each object does not matter.

Example

Systems that allow users to define their own application flows and layouts often have a need to keep track of large numbers of fields, pages, and other items that are almost identical to each other. By making these items into flyweights all instances of each object can share the intrinsic state while keeping the extrinsic state separate. The intrinsic state would store the shared properties, such as how a textbox looks, how much data it can hold, and what events it exposes. The extrinsic state would store the unshared properties, such as where the item belongs, how to react to a user click, and how to handle events.

PROXY — Object Structural



Purpose

Allows for object level access control by acting as a pass through entity or a placeholder object.

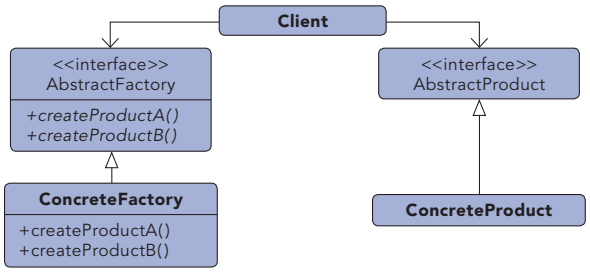
Use When

- The object being represented is external to the system.
- Objects need to be created on demand.
- Access control for the original object is required.
- Added functionality is required when an object is accessed.

Example

Ledger applications often provide a way for users to reconcile their bank statements with their ledger data on demand, automating much of the process. The actual operation of communicating with a third party is a relatively expensive operation that should be limited. By using a proxy to represent the communications object we can limit the number of times or the intervals the communication is invoked. In addition, we can wrap the complex instantiation of the communication object inside the proxy class, decoupling calling code from the implementation details.

ABSTRACT FACTORY — Object Creational



Purpose

Provide an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.

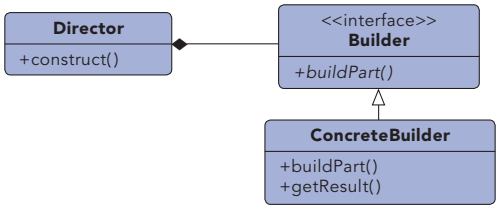
Use When

- The creation of objects should be independent of the system utilizing them.
- Systems should be capable of using multiple families of objects.
- Families of objects must be used together.
- Libraries must be published without exposing implementation details.
- Concrete classes should be decoupled from clients.

Example

Email editors will allow for editing in multiple formats including plain text, rich text, and HTML. Depending on the format being used, different objects will need to be created. If the message is plain text then there could be a body object that represented just plain text and an attachment object that simply encrypted the attachment into Base64. If the message is HTML then the body object would represent HTML encoded text and the attachment object would allow for inline representation and a standard attachment. By utilizing an abstract factory for creation we can then ensure that the appropriate object sets are created based upon the style of email that is being sent.

BUILDER — Object Creational



Purpose

Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

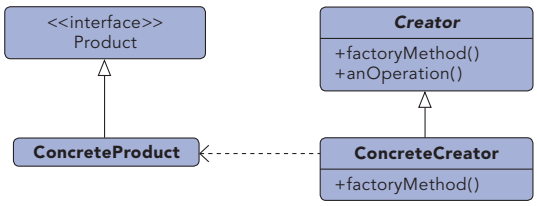
Use When

- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

Example

A file transfer application could possibly use many different protocols to send files and the actual transfer object that will be created will be directly dependent on the chosen protocol. Using a builder we can determine the right builder to use to instantiate the right object. If the setting is FTP then the FTP builder would be used when creating the object.

FACTORY METHOD — Object Creational



Purpose

Exposes a method for creating objects, allowing subclasses to control the actual creation process.

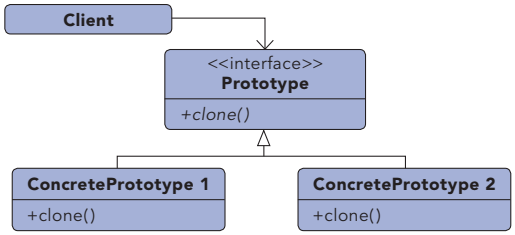
Use When

- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

Example

Many applications have some form of user and group structure for security. When the application needs to create a user it will typically delegate the creation of the user to multiple user implementations. The parent user object will handle most operations for each user but the subclasses will define the factory method that handles the distinctions in the creation of each type of user. A system may have AdminUser and StandardUser objects each of which extend the User object. The AdminUser object may perform some extra tasks to ensure access while the StandardUser may do the same to limit access.

PROTOTYPE — Object Creational



Purpose

Create objects based upon a template of an existing objects through cloning.

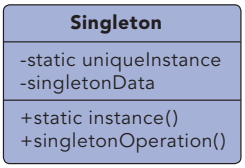
Use When

- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

Example

Rates processing engines often require the lookup of many different configuration values, making the initialization of the engine a relatively expensive process. When multiple instances of the engine is needed, say for importing data in a multi-threaded manner, the expense of initializing many engines is high. By utilizing the prototype pattern we can ensure that only a single copy of the engine has to be initialized then simply clone the engine to create a duplicate of the already initialized object. The added benefit of this is that the clones can be streamlined to only include relevant data for their situation.

SINGLETON — Object Creational



Purpose

Ensures that only one instance of a class is allowed within a system.

Use When

- Exactly one instance of a class is required.
- Controlled access to a single object is necessary.

Example

Most languages provide some sort of system or environment object that allows the language to interact with the native operating system. Since the application is physically running on only one operating system there is only ever a need for a single instance of this system object. The singleton pattern would be implemented by the language runtime to ensure that only a single copy of the system object is created and to ensure only appropriate processes are allowed access to it.

ABOUT THE AUTHOR

Jason McDonald

The product of two computer programmers, Jason McDonald wrote his first application in BASIC while still in elementary school and has been heavily involved in software ever since. He began his software career when he found himself automating large portions of one of his first jobs. Finding his true calling, he quit the position and began working as a software engineer for various small companies where he was responsible for all aspects of applications, from initial design to support. He has roughly 11 years of experience in the software industry and many additional years of personal software experience during which he has done everything from coding to architecture to leading and managing teams of engineers. Through his various positions he has been exposed to design patterns and other architectural concepts for years. Jason is the founder of the Charleston SC Java Users Group and is currently working to help found a Charleston chapter of the International Association of Software Architects.

Personal Blog: http://www.mcdonaldland.info/
Projects: Charleston SC Java Users Group

RECOMMENDED BOOK

Capturing a wealth of experience about the design of object-oriented software, four top-notch designers present a catalog of simple and succinct solutions to commonly occurring design problems. Previously undocumented, these 23 patterns allow designers to create more flexible, elegant, and ultimately reusable designs without having to rediscover the design solutions themselves.

BUY NOW
books.dzone.com/books/designpatterns

Get More FREE Refcardz. Visit refcardz.com now!

Upcoming Refcardz:

Core Seam
Core CSS: Part III
Hibernate Search
Equinox
EMF
XML
JSP Expression Language
ALM Best Practices
HTML and XHTML

Available:

Essential Ruby
Essential MySQL
JUnit and EasyMock
Getting Started with MyEclipse
Spring Annotations
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET
Very First Steps in Flex
C#
Groovy
NetBeans IDE 6.1 Java Editor
RSS and Atom
GlassFish Application Server
Silverlight 2

Visit refcardz.com for a complete listing of available Refcardz.

Design Patterns
Published June 2008

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more.
"DZone is a developer's dream," says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513
888.678.0399
919.678.0300
Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-10-3
ISBN-10: 1-934238-10-4
50795
9 781934 238103
$7.95

Design-Patterns-Katalog

Erzeugungsmuster (Creational Patterns)

***Abstract Factory* (*Abstrakte Fabrik*, siehe Abschnitt 3.1)**

Bereitstellung einer Schnittstelle zum Erzeugen verwandter oder voneinander abhängiger Objektfamilien ohne die Benennung ihrer konkreten Klassen.

***Builder* (*Erbauer*, siehe Abschnitt 3.2)**

Getrennte Handhabung der Erzeugungs- und Darstellungsmechanismen komplexer Objekte zwecks Generierung verschiedener Repräsentationen in einem einzigen Erzeugungsprozess.

***Factory Method* (*Fabrikmethode*, siehe Abschnitt 3.3)**

Definition einer Schnittstelle zur Objekterzeugung, wobei die Bestimmung der zu instanziierenden Klasse den Unterklassen überlassen bleibt. Das Design Pattern *Factory Method (Fabrikmethode)* gestattet einer Klasse, die Instanziierung an Unterklassen zu delegieren.

***Prototype* (*Prototyp*, siehe Abschnitt 3.4)**

Spezifikation der zu erzeugenden Objekttypen mittels einer prototypischen Instanz und Erzeugung neuer Objekte durch Kopieren dieses Prototyps.

***Singleton* (*Singleton*, siehe Abschnitt 3.5)**

Sicherstellung der Existenz nur einer einzigen Klasseninstanz sowie Bereitstellung eines globalen Zugriffspunkts für diese Instanz.

Strukturmuster (Structural Patterns)

Adapter (*Adapter*, siehe Abschnitt 4.1)

Anpassung der Schnittstelle einer Klasse an ein anderes von den Clients erwartetes Interface. Das Design Pattern *Adapter (Adapter)* ermöglicht die Zusammenarbeit von Klassen, die ansonsten aufgrund der Inkompatibilität ihrer Schnittstellen nicht dazu in der Lage wären.

Bridge (*Brücke*, siehe Abschnitt 4.2)

Entkopplung einer Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

Composite (*Kompositum*, siehe Abschnitt 4.3)

Komposition von Objekten in Baumstrukturen zur Abbildung von Teil-Ganzes-Hierarchien. Das Design Pattern *Composite (Kompositum)* gestattet den Clients einen einheitlichen Umgang sowohl mit individuellen Objekten als auch mit Objektkompositionen.

Decorator (*Dekorierer*, siehe Abschnitt 4.4)

Dynamische Erweiterung der Funktionalität eines Objekts. `Decorator`-Objekte stellen hinsichtlich der Ergänzung einer Klasse um weitere Zuständigkeiten eine flexible Alternative zur Unterklassenbildung dar.

Facade (*Fassade*, siehe Abschnitt 4.5)

Bereitstellung einer einheitlichen Schnittstelle zu einem Schnittstellensatz in einem Subsystem. Das Design Pattern *Facade (Fassade)* definiert eine Schnittstelle höherer Ebene, die die Nutzung des Subsystems vereinfacht.

***Flyweight* (*Fliegengewicht*, siehe Abschnitt 4.6)**

Gemeinsame Nutzung feingranularer Objekte, um sie auch in großer Anzahl effizient nutzen zu können.

***Proxy* (*Proxy*, siehe Abschnitt 4.7)**

Bereitstellung eines vorgelagerten Stellvertreterobjekts bzw. eines Platzhalters zwecks Zugriffssteuerung eines Objekts.

Verhaltensmuster (Behavioral Patterns)

***Chain of Responsibility* (*Zuständigkeitskette*, siehe Abschnitt 5.1)**

Unterbindung der Kopplung eines Request-Auslösers mit seinem Empfänger, indem mehr als ein Objekt in die Lage versetzt wird, den Request zu bearbeiten. Die empfangenden Objekte werden miteinander verkettet und der Request wird dann so lange entlang dieser Kette weitergeleitet, bis er von einem Objekt angenommen und abgearbeitet wird.

***Command* (*Befehl*, siehe Abschnitt 5.2)**

Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiedenen Requests, Warteschlangen- oder Logging-Operationen sowie das Rückgängigmachen von Operationen zu ermöglichen.

***Interpreter* (*Interpreter*, siehe Abschnitt 5.3)**

Definition einer Repräsentation für die Grammatik einer gegebenen Sprache und Bereitstellung eines Interpreters, der sie nutzt, um in dieser Sprache verfasste Sätze zu interpretieren.

***Iterator* (*Iterator*, siehe Abschnitt 5.4)**

Bereitstellung eines sequenziellen Zugriffs auf die Elemente eines aggregierten Objekts, ohne dessen zugrunde liegende Struktur offenzulegen.

***Mediator* (*Vermittler*, siehe Abschnitt 5.5)**

Definition eines Objekts, das die Interaktionsweise eines Objektsatzes in sich kapselt. Das Design Pattern *Mediator (Vermittler)* begünstigt lose Kopplungen, indem es die explizite Referenzierung der Objekte untereinander unterbindet und so eine individuelle Steuerung ihrer Interaktionen ermöglicht.

***Memento* (*Memento*, siehe Abschnitt 5.6)**

Erfassung und Externalisierung des internen Zustands eines Objekts, ohne dessen Kapselung zu beeinträchtigen, so dass es später wieder in diesen Zustand zurückversetzt werden kann.

***Observer* (*Beobachter*, siehe Abschnitt 5.7)**

Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Objekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.

***State* (*Zustand*, siehe Abschnitt 5.8)**

Anpassung der Verhaltensweise eines Objekts im Fall einer internen Zustandsänderung, so dass es den Anschein hat, als hätte es seine Klasse gewechselt.

***Strategy* (*Strategie*, siehe Abschnitt 5.9)**

Definition einer Familie von einzeln gekapselten, austauschbaren Algorithmen. Das Design Pattern *Strategy (Strategie)* ermöglicht eine variable und von den Clients unabhängige Nutzung des Algorithmus.

***Template Method* (*Schablonenmethode*, siehe Abschnitt 5.10)**

Definition der Grundstruktur eines Algorithmus in einer Operation sowie Delegation einiger Ablaufschritte an Unterklassen. Das Design Pattern *Template Method (Schablonenmethode)* ermöglicht den Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne dessen grundlegende Struktur zu verändern.

***Visitor* (*Besucher*, siehe Abschnitt 5.11)**

Darstellung einer auf die Elemente einer Objektstruktur auszuführenden Operation. Das Design Pattern *Visitor (Besucher)* ermöglicht die Definition einer neuen Operation, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

6.1. Example

Informally, use cases are *text stories* of some actor using a system to meet goals. Here is an example *brief format* use case:

> **Process Sale:** A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Notice that ***use cases are not diagrams, they are text***. Focusing on secondary-value UML use case diagrams rather than the important use case text is a common mistake for use case novices.

UML use case diagrams p. *89*

Use cases often need to be more detailed or structured than this example, but the essence is discovering and recording functional requirements by writing stories of using a system to fulfill user goals; that is, *cases of use*.[1] It isn't supposed to be a difficult idea, although it's often difficult to discover what's needed and write it well.

[1] The original term in Swedish literally translates as "usage case."

6.2. Definition: What are Actors, Scenarios, and Use Cases?

First, some informal definitions: an **actor** is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

A **scenario** is a specific sequence of actions and interactions between actors and the system; it is also called a **use case instance**. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.

Informally then, a **use case** is a collection of related success and failure scenarios that describe an actor using a system to support a goal. For example, here is a *casual format* use case with alternate scenarios:

> **Handle Returns**
>
> *Main Success Scenario*: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...
>
> *Alternate Scenarios*:
>
> If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.
>
> If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).
>
> If the system detects failure to communicate with the external accounting system, ...

Now that scenarios (use case instances) are defined, an alternate, but similar definition of a use case provided by the RUP will make better sense:

> A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor [RUP].

6.7. Notation: What are Three Common Use Case Formats?

Use cases can be written in different formats and levels of formality:

- **brief** Terse one-paragraph summary, usually of the main success scenario. The prior *Process Sale* example was brief.

example p. 63

  - When? During early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.

- **casual** Informal paragraph format. Multiple paragraphs that cover various scenarios. The prior *Handle Returns* example was casual.

example p. 63

  - When? As above.

- **fully dressed** All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

example p. 68

  - When? After many use cases have been identified and written in a brief format, then during the first requirements workshop a few (such as 10%) of the architecturally significant and high-value use cases are written in detail.

more on timing of writing use cases p. 95

6.8. Example: Process Sale, Fully Dressed Style

Fully dressed use cases show more detail and are structured; they dig deeper.

In iterative and evolutionary UP requirements analysis, 10% of the critical use cases would be written this way during the first requirements workshop. Then design and programming starts on the most architecturally significant use cases or scenarios from that 10% set.

Various format templates are available for detailed use cases. Probably the most widely used and shared format, since the early 1990s, is the template available on the Web at *alistair.cockburn.us*, created by Alistair Cockburn, the author of the most popular book and approach to use-case modeling. The following example illustrates this style.

Main Success Scenario and Extensions are the two major sections

First, here's the template:

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Here's an example, based on the template.

Please note that this is the book's primary case study example of a detailed use case; it shows many common elements and issues.

It probably shows much more than you ever wanted to know about a POS system! But, it's for a real POS, and shows the ability of use cases to capture complex real-world requirements, and deeply branching scenarios.

Use Case UC1: Process Sale

Scope: NextGen POS application

Level: user goal

Primary Actor: Cashier

Stakeholders and Interests:

- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.

- Salesperson: Wants sales commissions updated.

- Customer: Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.

- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.

- Manager: Wants to be able to quickly perform override operations, and easily debug Cashier problems.

- Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.

- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions: Cashier is identified and authenticated.

Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):

1. Customer arrives at 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. Price calculated from a set of price rules.

   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.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

* a. At any time, Manager requests an override operation:

1. System enters Manager-authorized mode.

- Manager or Cashier performs one Manager-mode operation. e.g., cash balance change, resume a suspended sale on another register, void a sale, etc.
- System reverts to Cashier-authorized mode.

* b. At any time, System fails:

To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.

- System reconstructs prior state.

  2a. System detects anomalies preventing recovery:

  1. System signals error to the Cashier, records the error, and enters a clean state.

- Cashier starts a new sale.

1a. Customer or Manager indicate to resume a suspended sale.

1. Cashier performs resume operation, and enters the ID to retrieve the sale.
2. System displays the state of the resumed sale, with subtotal.

    2a. Sale not found.

    1. System signals error to the Cashier.

    - Cashier probably starts new sale and re-enters all items.

- Cashier continues with sale (probably entering more items or handling payment).

2-4a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples)

1. Cashier verifies, and then enters tax-exempt status code.
2. System records status (which it will use during tax calculations)

3a. Invalid item ID (not found in system):

1. System signals error and rejects entry.
2. Cashier responds to the error:

    2a. There is a human-readable item ID (e.g., a numeric UPC):

    1. Cashier manually enters the item ID.

    - System displays description and price.

        2a. Invalid item ID: System signals error. Cashier tries alternate method.

    2b. There is no item ID, but there is a price on the tag:

    1. Cashier asks Manager to perform an override operation.

    - Managers performs override.
    - Cashier indicates manual price entry, enters price, and requests standard taxation for this amount (because there is no product information, the tax engine can't otherwise deduce how to tax it)

2c. Cashier performs Find Product Help to obtain true item ID and price.

2d. Otherwise, Cashier asks an employee for the true item ID or price, and does either manual ID or manual price entry (see above).

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

3c. Item requires manual category and price entry (such as flowers or cards with a price on them):

1. Cashier enters special manual category code, plus the price.

3-6a: Customer asks Cashier to remove (i.e., void) an item from the purchase:

This is only legal if the item value is less than the void limit for Cashiers, otherwise a Manager override is needed.

1. Cashier enters item identifier for removal from sale.
2. System removes item and displays updated running total.

    2a. Item price exceeds void limit for Cashiers:

    1. System signals error, and suggests Manager override.

    - Cashier requests Manager override, gets it, and repeats operation.

3-6b. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:

1. System records sale so that it is available for retrieval on any POS register.
2. System presents a "suspend receipt" that includes the line items, and a sale ID used to retrieve and resume the sale.

4a. The system supplied item price is not wanted (e.g., Customer complained about something

and is offered a lower price):

1. Cashier requests approval from Manager.
2. Manager performs override operation.
3. Cashier enters manual override price.
4. System presents new price.

5a. System detects failure to communicate with external tax calculation system service:

1. System restarts the service on the POS node, and continues.

    1a. System detects that the service does not restart.

    1. System signals error.

    - Cashier may manually calculate and enter the tax, or cancel the sale.

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):

1. Cashier signals discount request.
2. Cashier enters Customer identification.
3. System presents discount total, based on discount rules.

5c. Customer says they have credit in their account, to apply to the sale:

1. Cashier signals credit request.
2. Cashier enters Customer identification.
3. Systems applies credit up to price=0, and reduces remaining credit.

6a. Customer says they intended to pay by cash but don't have enough cash:

1. Cashier asks for alternate payment method.

    1a. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

7b. Paying by credit:

1. Customer enters their credit account information.
2. System displays their payment for verification.
3. Cashier confirms.

    3a. Cashier cancels payment step:

    1. System reverts to "item entry" mode.

- System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.

    4a. System detects failure to collaborate with external system:

    1. System signals error to Cashier.

    - Cashier asks Customer for alternate payment.

- System receives payment approval, signals approval to Cashier, and releases cash drawer (to insert signed credit payment receipt).

    5a. System receives payment denial:

    1. System signals denial to Cashier.

    - Cashier asks Customer for alternate payment.

5b. Timeout waiting for response.

1. System signals timeout to Cashier.
2. Cashier may try again, or ask Customer for alternate payment.

- System records the credit payment, which includes the payment approval.
- System presents credit payment signature input mechanism.
- Cashier asks Customer for a credit payment signature. Customer enters signature.
- If signature on paper receipt, Cashier places receipt in cash drawer and closes it.

7c. Paying by check...

7d. Paying by debit...

7e. Cashier cancels payment step:

1. System reverts to "item entry" mode.

7f. Customer presents coupons:

1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.

    1a. Coupon entered is not for any purchased item:

    1. System signals error to Cashier.

9a. There are product rebates:

1. System presents the rebate forms and rebate receipts for each item with a rebate.

9b. Customer requests gift receipt (no prices visible):

1. Cashier requests gift receipt and System presents it.

9c. Printer out of paper.

1. If System can detect the fault, will signal the problem.
2. Cashier replaces paper.
3. Cashier requests another receipt.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.

- Credit authorization response within 30 seconds 90% of the time.

- Somehow, we want robust recovery when access to remote services such the inventory system is failing.

- Language internationalization on the text displayed.

- Pluggable business rules to be insertable at steps 3 and 7.

- ...

Technology and Data Variations List:

*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.

3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.

3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

7a. Credit account information entered by card reader or keyboard.

7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

- What are the tax law variations?

- Explore the remote service recovery issue.

- What customization is needed for different businesses?

- Must a cashier take their cash drawer when they log out?

- Can the customer directly use the card reader, or does the cashier have to do it?

This use case is illustrative rather than exhaustive (although it is based on a real POS system's requirementsdeveloped with an OO design in Java). Nevertheless, there is enough detail and complexity here to offer a realistic sense that a fully dressed use case can record many requirement details. This example will serve well as a model for many use case problems.

6.13. Guideline: Write Black-Box Use Cases

Black-box use cases are the most common and recommended kind; they do not describe the internal workings of the system, its components, or design. Rather, the system is described as having *responsibilities*, which is a common unifying metaphorical theme in object-oriented thinkingsoftware elements have responsibilities and collaborate with other elements that have responsibilities.

By defining system responsibilities with black-box use cases, one can specify *what* the system must do (the behavior or functional requirements) without deciding *how* it will do it (the design). Indeed, the definition of "analysis" versus "design" is sometimes summarized as "what" versus "how." This is an important theme in good software development: During requirements analysis avoid making "how" decisions, and specify the external behavior for the system, as a black box. Later, during design, create a solution that meets the specification.

Black-box style	Not
The system records the sale.	The system writes the sale to a database. ..or (even worse): The system generates a SQL INSERT statement for the sale...

6.16. Guideline: What Tests Can Help Find Useful Use Cases?

Which of these is a valid use case?

- Negotiate a Supplier Contract
- Handle Returns
- Log In
- Move Piece on Game Board

An argument can be made that all of these are use cases *at different levels*, depending on the system boundary, actors, and goals.

But rather than asking in general, "What is a valid use case?", a more practical question is: "What is a useful level to express use cases for application requirements analysis?" There are several rules of thumb, including:

- The Boss Test
- The EBP Test
- The Size Test

The Boss Test

Your boss asks, "What have you been doing all day?" You reply: "Logging in!" Is your boss happy?

If not, the use case fails the Boss Test, which implies it is not strongly related to achieving results of measurable value. It may be a use case at some low goal level, but not the desirable level of focus for requirements analysis.

That doesn't mean to always ignore boss-test-failing use cases. User authentication may fail the boss test, but may be important and difficult.

The EBP Test

An **Elementary Business Process** (**EBP**) is a term from the business process engineering field,[5] defined as:

> [5] EBP is similar to the term **user task** in usability engineering, although the meaning is less strict in that domain.
>
> A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state, e.g., Approve Credit or Price Order [original source lost].

Focus on use cases that reflect EBPs.

The EBP Test is similar to the Boss Test, especially in terms of the measurable business value qualification.

The definition can be taken too literally: Does a use case fail as an EBP if two people are required, or if a person has to walk around? Probably not, but the feel of the definition is about right. It's not a single small step like "delete a line item" or "print the document." Rather, the main success scenario is probably five or ten steps. It doesn't take days and multiple sessions, like "negotiate a supplier contract"; it is a task done during a single session. It is probably between a few minutes and an hour in length. As with the UP's definition, it emphasizes adding observable or measurable business value, and it comes to a resolution in which the system and data are in a stable and consistent state.

The Size Test

A use case is very seldom a single action or step; rather, a use case typically contains many steps, and in the fully dressed format will often require 310 pages of text. A common mistake in use case modeling is to define just a single step within a series of related steps as a use case by itself, such as defining a use case called *Enter an Item ID*. You can see a hint of the error by its small sizethe use case name will wrongly suggest just one step within a larger series of steps, and if you imagine the length of its fully dressed text, it would be extremely short.

Example: Applying the Tests

- Negotiate a Supplier Contract
    - Much broader and longer than an EBP. Could be modeled as a *business* use case, rather than a system use case.
- Handle Returns
    - OK with the boss. Seems like an EBP. Size is good.
- Log In
    - Boss not happy if this is all you do all day!
- Move Piece on Game Board
    - Single stepfails the size test.

Reasonable Violations of the Tests

Although the majority of use cases identified and analyzed for an application should satisfy the tests, exceptions are common.

It is sometimes useful to write separate subfunction-level use cases representing subtasks or steps within a regular EBP-level use case. For example, a subtask or extension such as "paying by credit" may be repeated in several base use cases. If so, it is desirable to separate this into its own use case, even though it does not really satisfy the EBP and size tests, and link it to several base use cases, to avoid duplication of the text.

see the use case "include" relationship for more on linking subfunction use cases p. 494

Authenticate User may not pass the Boss test, but be complex enough to warrant careful analysis, such as for a "single sign-on" feature.

PREVIOUS NEXT

5.4. What are the Types and Categories of Requirements?

In the UP, requirements are categorized according to the FURPS+ model [Grady92], a useful mnemonic with the following meaning:[1]

[1] There are several systems of requirements categorization and quality attributes published in books and by standards organizations, such as ISO 9126 (which is similar to the FURPS+ list), and several from the Software Engineering Institute (SEI); any can be used on a UP project.

- **Functional** features, capabilities, security.
- **Usability** human factors, help, documentation.
- **Reliability** frequency of failure, recoverability, predictability.
- **Performance** response times, throughput, accuracy, availability, resource usage.
- **Supportability** adaptability, maintainability, internationalization, configurability.

The "+" in FURPS+ indicates ancillary and sub-factors, such as:

- **Implementation** resource limitations, languages and tools, hardware, ...
- **Interface** constraints imposed by interfacing with external systems.
- **Operations** system management in its operational setting.
- **Packaging** for example, a physical box.
- **Legal** licensing and so forth.

It is helpful to use FURPS+ categories (or some categorization scheme) as a checklist for requirements coverage, to reduce the risk of not considering some important facet of the system.

Some of these requirements are collectively called the **quality attributes, quality requirements**, or the "-ilities" of a system. These include usability, reliability, performance, and supportability. In common usage, requirements are categorized as **functional** (behavioral) or **non-functional** (everything else); some dislike this broad generalization [BCK98], but it is very widely used.

As we shall see when exploring architectural analysis, the quality attributes have a strong influence on the architecture of a system. For example, a high-performance, high-reliability requirement will influence the choice of software and hardware components, and their configuration.

architectural analysis p. 541

4. Zusätzliche Spezifikation

Folgende Anforderungen wurden gemäss FURPS+ definiert.

Functionality

Folgende weitere funktionale Anforderungen wurden identifiziert:

- Alle Fehler und Aktionen werden über einen gewissen Zeitraum aufgezeichnet, um eine spätere Analyse zu ermöglichen.
- Die Übertragung der geteilten Einkaufslisten erfolgt über den bestehenden Service „Firebase Cloud Messaging“.
- *MealManager* wird mit Java entwickelt.
- *MealManager* muss auf Android Systemen ab Version 6.x (Marshmallow) funktionieren.

Usability

- *MealManager* muss für Smartphones per Touch einfach bedienbar sein, dass bedeutet z.B. grosse Knöpfe und Eingabefelder.
- *MealManager* muss einfach bedienbar sein.

Reliability

- *MealManager* muss immer verfügbar sein.
- Wenn der Kunde eine stabile Internetverbindung hat, muss er Einkaufslisten teilen und empfangen können.
- Die Daten (Einkaufsliste, Rezepte, usw.) müssen nach einem Absturz weiterhin verfügbar sein.
- *MealManager* muss im Offline-Modus funktionieren. Ausnahme Einkaufsliste teilen/empfangen.

Performance

- *MealManager* muss innert 2 Sekunden auf einen Knopfdruck reagieren.
- *MealManager* muss Einkaufslisten innerhalb von 5 Sekunden übertragen.

Supportability

- Einzelne Komponenten müssen einfach erweiterbar sein, per Update im Play-Store.

+ Implementation Restrictions, Interfaces, Operations, Packaging

- Die Designvorgaben von Android Version 6.0 müssen eingehalten werden.
- *MealManager* wird aus dem Google Play Store heruntergeladen.