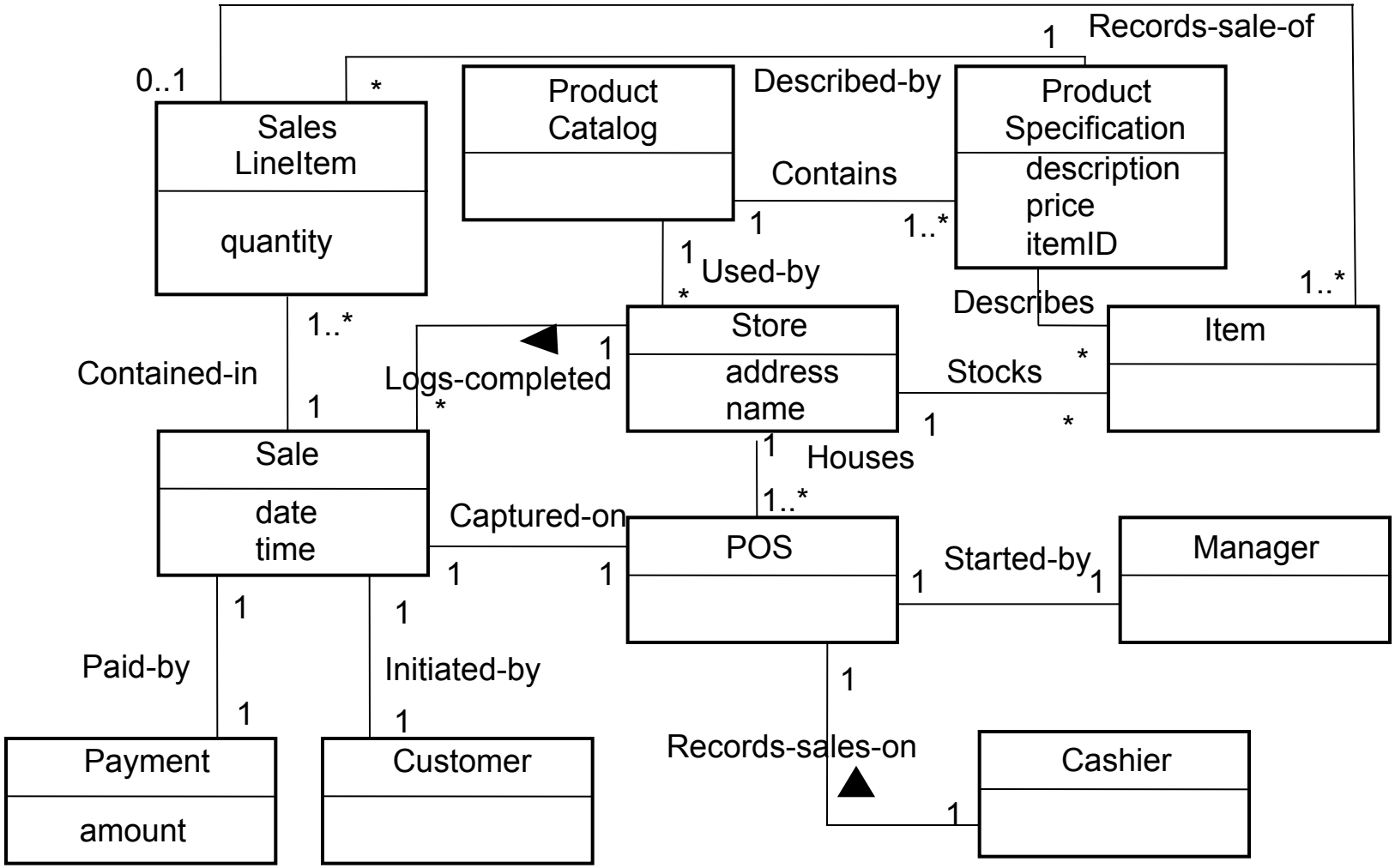
GRASP*: Návrh Objektů se Zodpovědnostmi

* Genenal Responsibility Assignment Software Patterns

Doménový Model



Zodpovědnosti a Metody

- Záměrem objektového návrhu je identifikace tříd a objektů, rozhodnutí které metody patří kam a kde a jak tyto objekty spolupracují.
- Zodpovědnosti mají vztah k závazku objektu ve smyslu jeho chování.
- Dva typy zodpovědností:
  - Doing:
    - Sám něco dělá (e.g. creating an object, doing a calculation)
    - Spouští akci na jiném objektu.
    - Kontroluje a koordinuje aktivity na jiných objektech.
  - Knowing:
    - Má znalost privátních zapouzdřených dat.
    - Má znalost o vztazích objektů.
    - Zná vše potřebné k derivaci, kalkulaci.

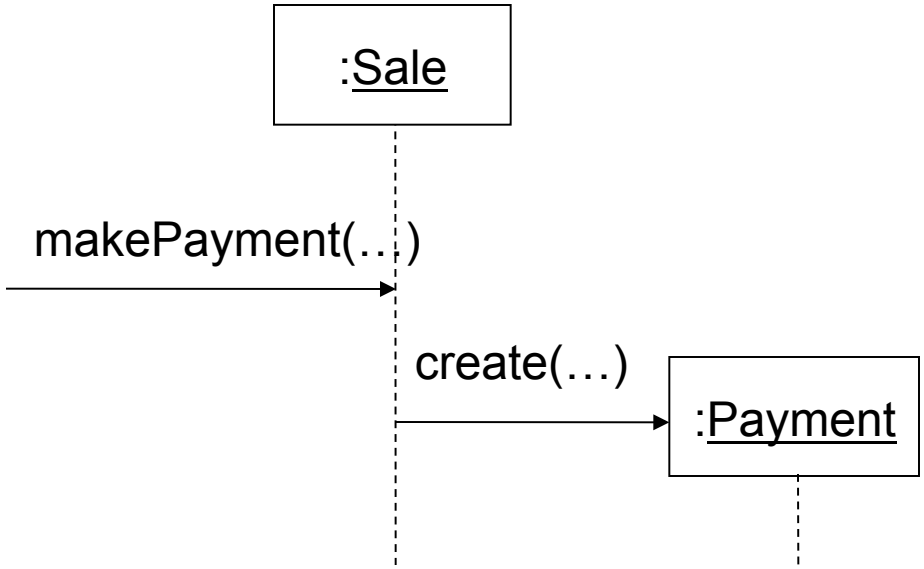
Zodpovědnosti a Metody

- Zodpovědnosti jsou přiřazeny třídám v objektovém návrhu. Např:
  - *"a Sale is responsible for creating SalesLineItems"* (doing)
  - *"a Sale is responsible for knowing its total"* (knowing)
- Zodpovědnosti s vztahem k "knowing" jsou často odvozeny z Doménového Modelu
  - (because of the attributes and associations it illustrates)

Zodpovědnosti a Metody

- Překlad zodpovědností do tříd a metod je ovlivněno granularitou zodpovědnosti.
  - For example, *"provide access to relational databases"* may involve dozens of classes and hundreds of methods, whereas *"create a Sale"* may involve only one or few methods.
- Zodpovědnost není to samé jako metoda, nýbrž metody jsou implementovány k naplnění zodpovědností.
- Metody buď pracují sami nebo spolupracují s dalšími metodami a objekty.

Zodpovědnosti a Diagramy Interakce



- V UML artefaktech, obecný kontext, kde jsou tyto zodpovědnosti (implemented as methods) uvažovány je při tvorbě diagramů interakce.
- *Sale* objekt má zodpovědnost k vytvoření *Payments*, skrze metodu *makePayment*.

Vzory

- Zdůrazníme tyto principy (expressed in patterns) jako průvodce voleb, kam přiřadit zodpovědnosti.
- Vzor je pojmenovaný popis problému a řešení, které může být aplikováno na nové kontexty; poskytuje radu, jak aplikovat řešení v různých situacích.
- Např:
  - Pattern name: Information Expert
  - Problem: What is the most basic principle by which to assign responsibilities to objects?
  - Solution: Assign a responsibility to the class that has the information needed to fulfil it.

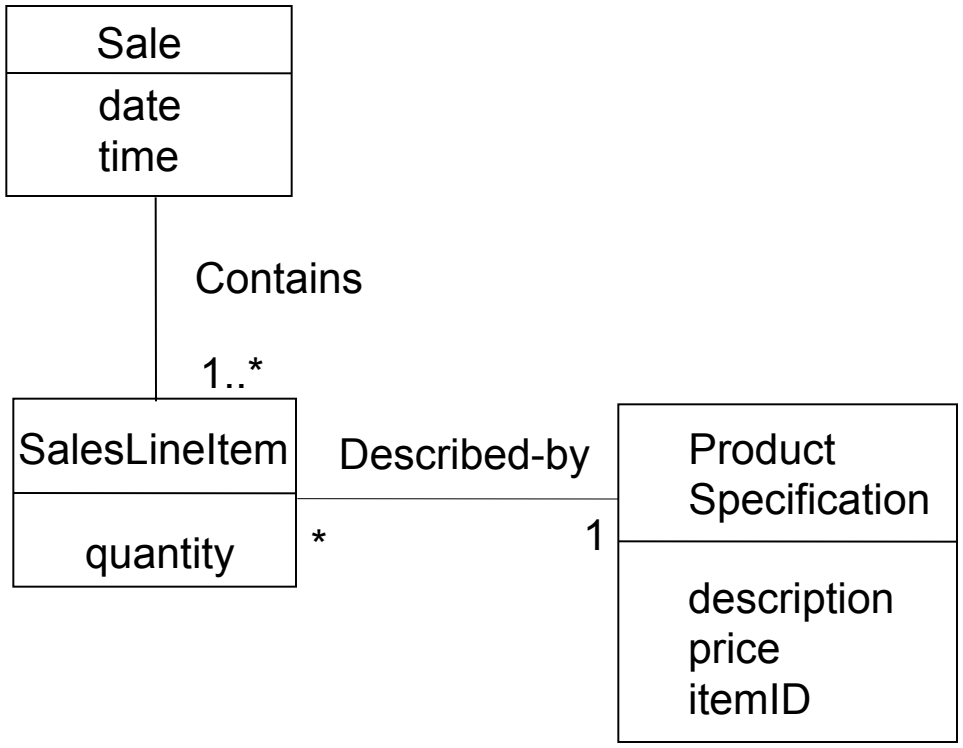
Information Expert (or Expert)

- Problem: co je obecný princip přiřazení zodpovědnosti objektům?
- Solution: přiřaď zodpovědnosti danému Expertu, tedy třídě co má informace k naplnění zodpovědnosti
- V NextGen POS aplikaci, kdo je zodpovědný za to znát celkový součet ceny (grand total) prodeje(sale)?
- Dle Information Expert se díváme po třídě která má informace k určení celkového součtu.

Information Expert (or Expert)

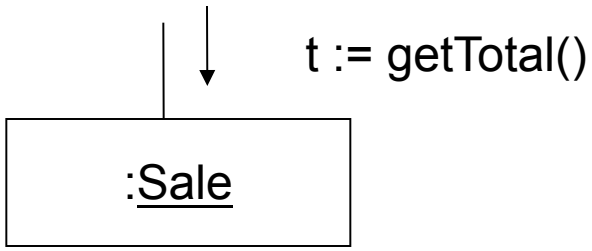
- Kam se budeme dívat abychom analyzovali třídy které mají dané informace? Do Domain Modelu, nebo Design Modelu..?
- Do obou! Můžeme předpokládat, že nemáme žádný či jen minimální Design Model. Určitě budeme hledat i v Domain Modelu daného information experta.

Information Expert (or Expert)

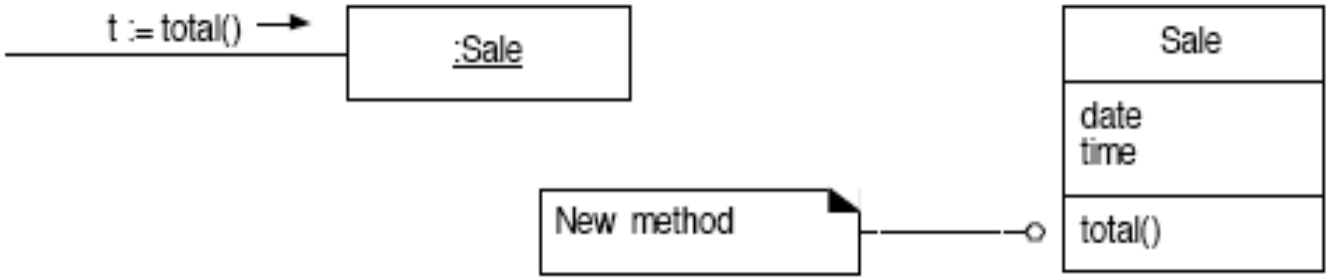


- Musíme znát o všech *SalesLineItem* instancích příslušných k *sale* a sečíst jejich mezisoučty (*subtotals*).
- *Sale* instance toto vše obsahuje,
  - i.e. it is an information expert for this responsibility.

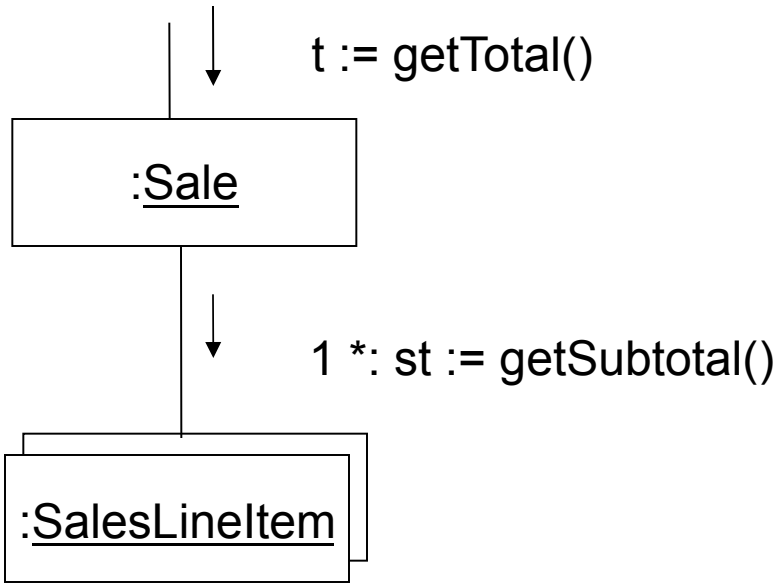
Information Expert (or Expert)



- Toto je částečný diagram interakce.

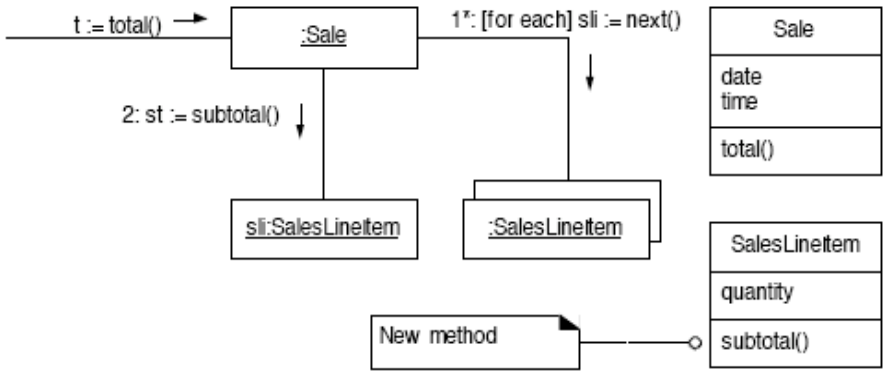


Information Expert (or Expert)



- Jaké informace jsou třeba k určení mezisoučtu položky (line item subtotal)?
  - quantity and price.
- *SalesLineItem* by mělo určit mezisoučet (subtotal).
- To znamená že *Sale* pošle getSubtotal() zprávu do každé *SalesLineItem* a sečte výsledky.

Information Expert (or Expert)

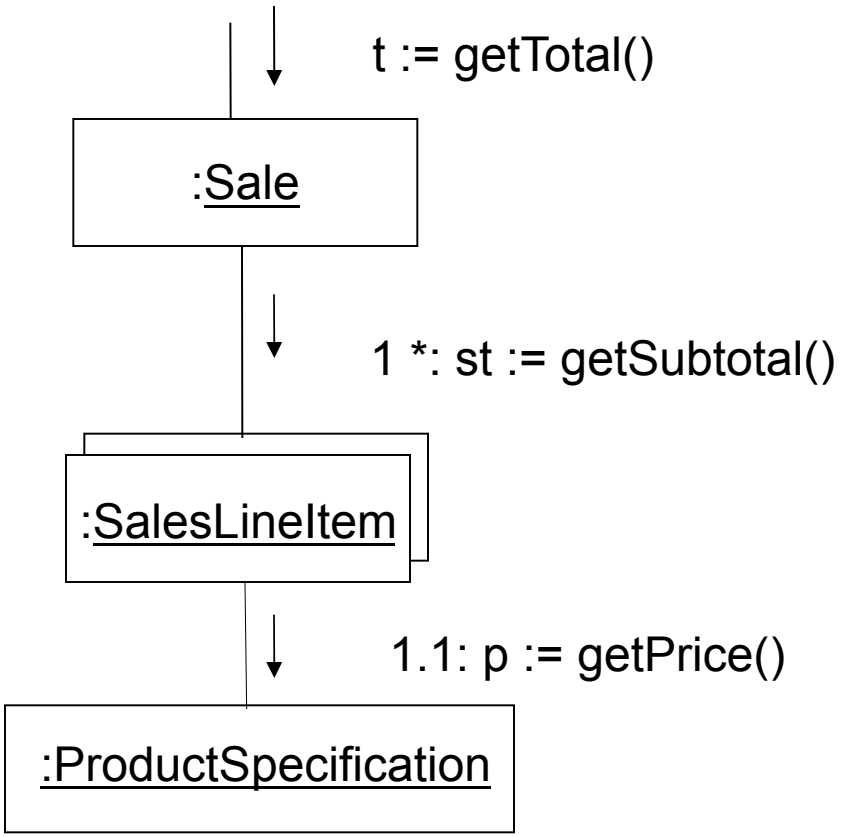


- Jaké informace jsou třeba k určení mezisoučtu položky (line item subtotal)?
  - quantity and price.

  SalesLineItem by mělo určit mezisoučet (subtotal).

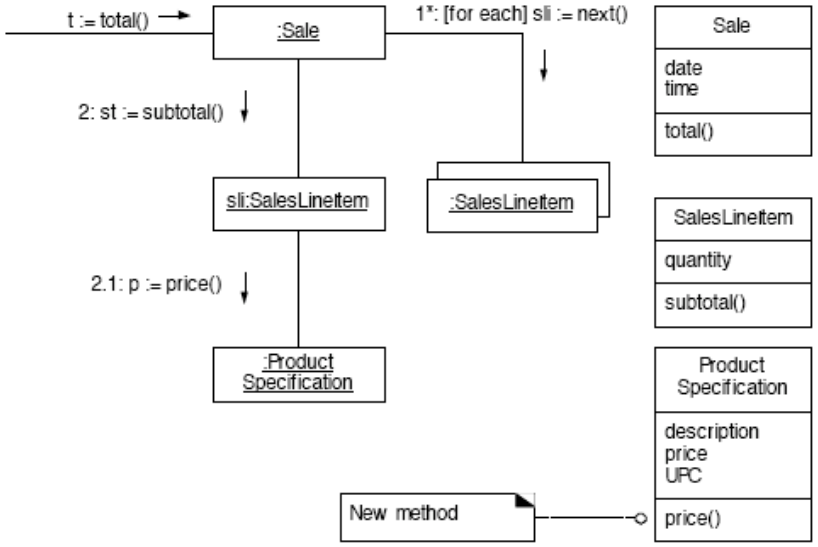
  To znamená že *Sale* pošle getSubtotal() zprávu do každé *SalesLineItem* a sečte výsledky.

Information Expert (or Expert)



- K naplnění zodpovědnosti "knowing and answering" jaký je mezisoučet (subtotal), *SalesLineItem* musí znát cenu produktu (price).
- *ProductSpecification* je information expert na poskytnutí své ceny.

Information Expert (or Expert)



- K naplnění zodpovědnosti "knowing and answering" jaký je mezisoučet (subtotal), *SalesLineItem* musí znát cenu produktu (price).
- *ProductSpecification* je information expert na poskytnutí své ceny.

Information Expert (or Expert)

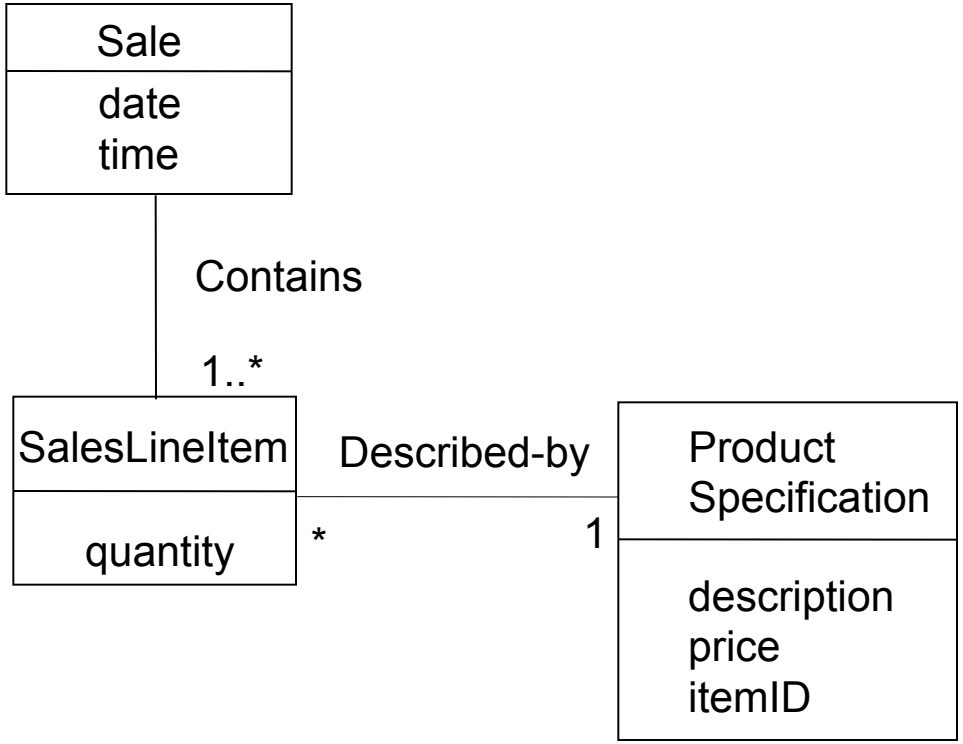
Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductSpecification	Knows product price

- K naplnění zodpovědnosti "knowing and answering" sale's total,
  - 3 zodpovědnosti byly přiřazeny 3 design třídám
- K naplnění zodpovědnosti je často třeba informací rozprostřených po mnoha třídách a objektech. To znamená že je mnoho "částečných expertů" kteří budou spolupracovat v daném úkolu.

Creator

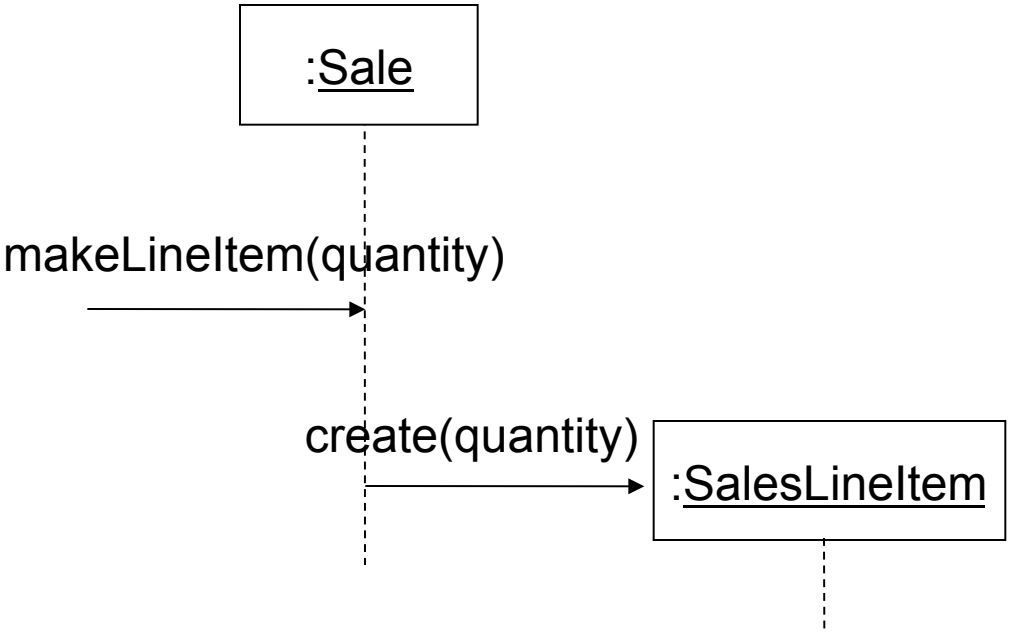
- Problem: Kdo by měl být zodpovědný za vytvoření nové instance nějaké třídy?
- Solution: Přiřaď třídě B zodpovědnost za vytvoření nové instance třídy A, pokud je jedna či více podmínek splněno:
  - B *aggregates* A objects.
  - B *contains* A objects.
  - B *records* instances of A objects.
  - B *has the initializing data* that will be passed to A when it is created
    - (thus B is an Expert with respect to creating A).

Creator

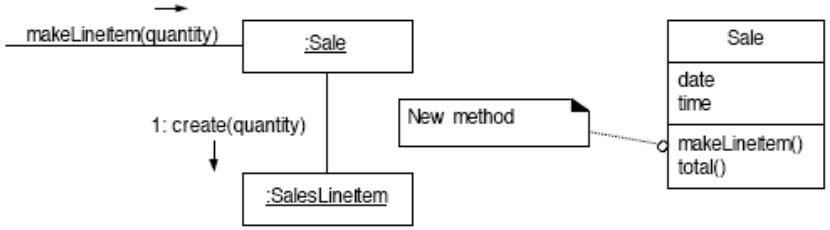


- V POS aplikaci, kdo je zodpovědný a vytvoření *SalesLineItem* instance?
- *Sale* obsahuje mnoho *SalesLineItem* objektů, Creator pattern tedy dle podmínek označuje *Sale* jako kandidáta.

Creator



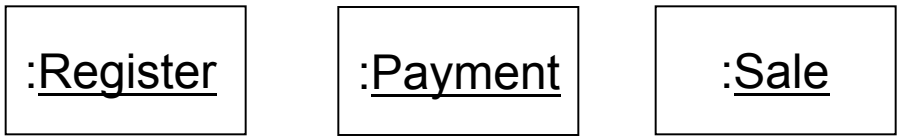
- Toto přiřazení zodpovědnosti požaduje že metoda *makeLineItem* bude definována nad *Sale*.



Low Coupling

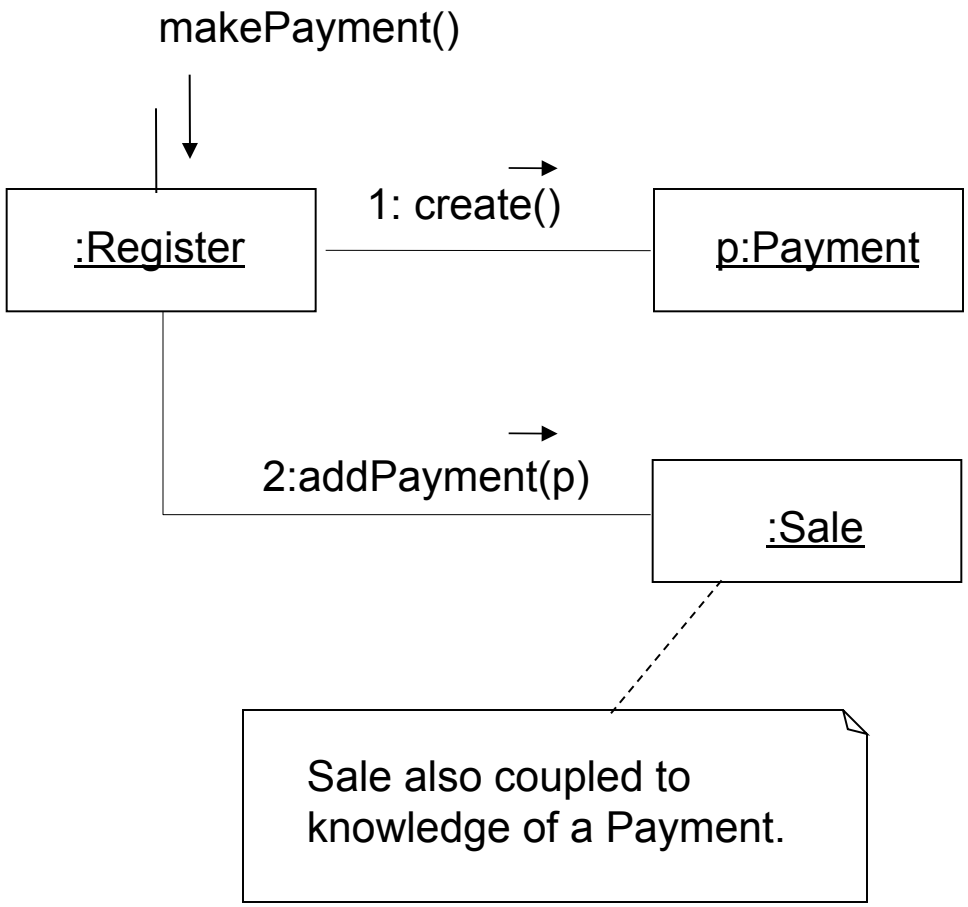
- Coupling: je míra jak silně je jeden element spojen, má znalost či spoléhá na dalších elementech.
- Třída mající high coupling je závislá na mnoha dalších třídách (libraries, tools).
- Problémy spojené s návrhem s high coupling:
  - Changes in related classes force local changes.
  - Harder to understand in isolation; need to understand other classes.
  - Harder to reuse because it requires additional presence of other classes.
- Problem: Jak podpořit co možná nejnižší závislosti, minimální dopad změn a zvýšit reuse/recyklaci?
- Solution: Přiřaď zodpovědnosti, tak že coupling zůstane malý.

Low Coupling



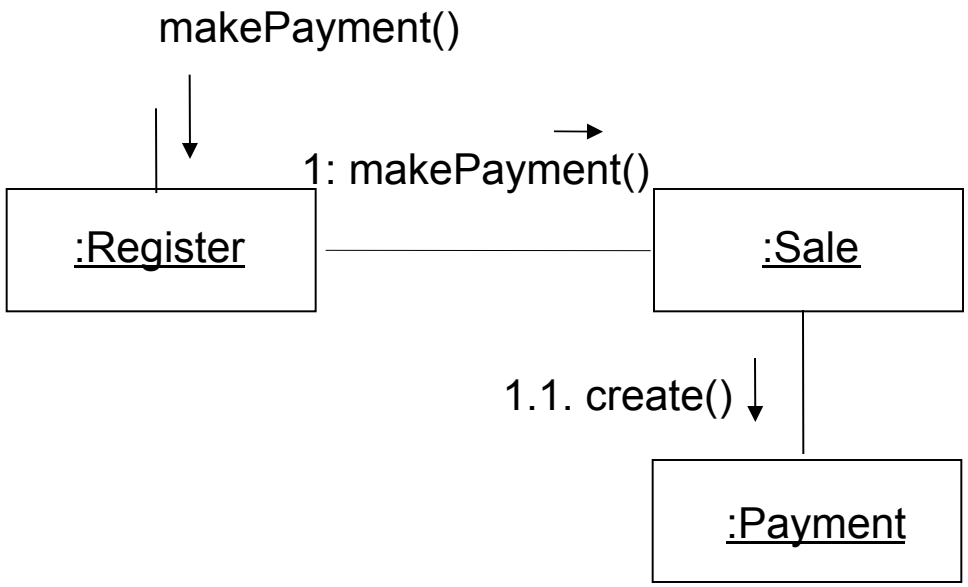
- Je třeba vytvořit instanci Payment a asociovat se Sale.
- Jaká třída by za to měla být zodpovědná?
- Dle Creatoru, Register je kandidát.

Low Coupling



- Register pak může poslat addPayment zprávu do Sale, a předat nový Payment jako parametr.
- Přiřazení zodpovědnosti coupluje Register class do znalosti Payment class.

Low Coupling



- Alternativa řešení je vytvořit Payment a asociovat se Sale.
  - No coupling between Register and Payment.

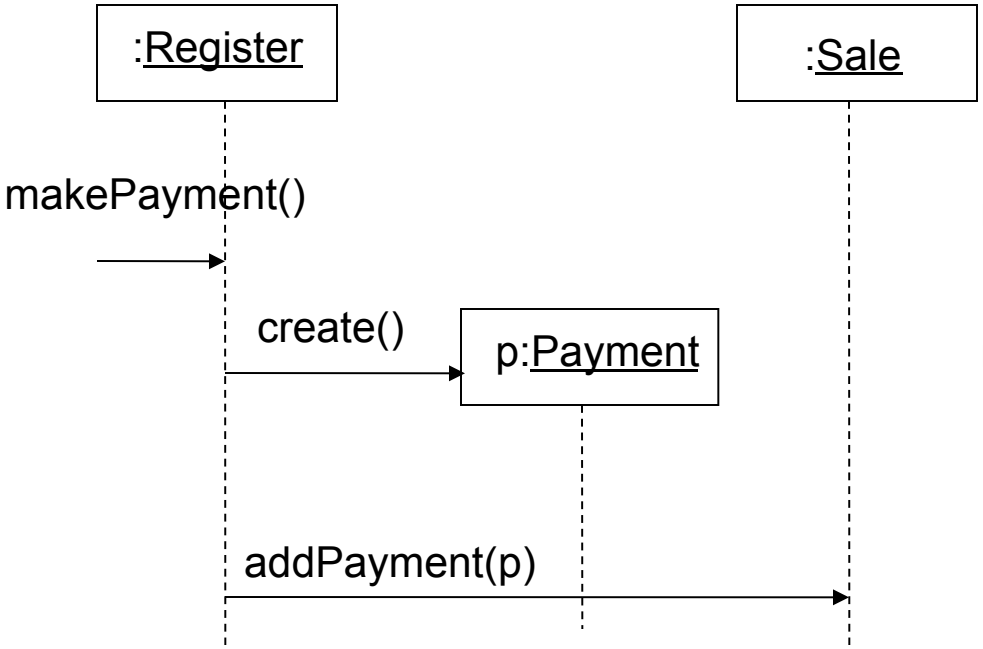
Low Coupling

- Kde se vyskytuje coupling:
  - Attributy: X has an attribute that refers to a Y instance.
  - Metody: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Podtřída: X is a subclass of Y.
  - Typy: X implements interface Y.
- Není specifické měření couplingu, ale obecně, třídy které jsou generické (šablony) a snadné na reuse/ recyklaci mají low coupling.
- Vždy bude existovat nějaký coupling mezi objekty, jinak by nemohla být spolupráce mezi objekty.

High Cohesion

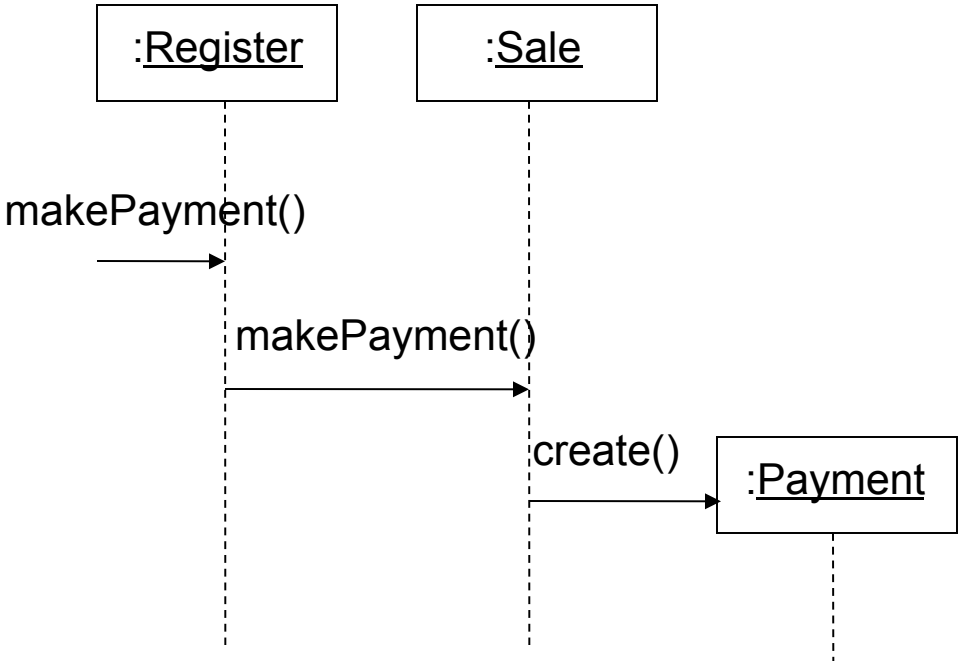
- Cohesion: je míra toho jak silně přísluší a účelově se zaměřují zodpovědnosti daného elementu.
- Třída s low cohesion dělá mnoho nesouvisejících aktivit nebo toho dělá až příliš.
- Problémy z důvodu návrhu s low cohesion:
  - Hard to understand.
  - Hard to reuse.
  - Hard to maintain.
  - Delicate, affected by change.
- Problem: Jak udržet složitost snadno ovladatelnou?
- Solution: Přiřaď zodpovědnosti tak, že cohesion zůstane high.

High Cohesion



- Potřebujeme vytvořit Payment instanci a asociovat ji se Sale.
  - Jaká třída za to má být zodpovědná?
- Dle Creatoru, Register je kandidát.
- Register se ale může stát “vyžraný”, pokud mu budeme přidávat další a další systémové operace.

High Cohesion



- Alternativní návrh deleguje zodpovědnost vytvoření *Payment* na *Sale*
  - Toto má higher cohesion v *Registeru*.
- Tedy tento návrh podporuje high cohesion a low coupling.

High Cohesion

- Scénáře ilustrující různé stupně funkční cohesion

1. Very low cohesion: třída zodpovědná za mnoho věcí v mnoha různých oblastech.
   - e.g.: a class responsible for interfacing with a data base and remote-procedure-calls.
2. Low cohesion: třída zodpovědná za složitý úkol ve funkční oblasti.
   - e.g.: a class responsible for interacting with a relational database.

High Cohesion

1. High cohesion: třída má mírnou zodpovědnost v jedné funkční oblasti a spolupracuje s dalšími třídami k naplnění úkolu.
   - e.g.: a class responsible for one section of interfacing with a data base.

- Hrubý odhad: třída s high cohesion má relativně málo metod, s hodně příbuznou funkčností, a nedělá toho moc. Spolupracuje a deleguje.

Low coupling VS. High Cohesion

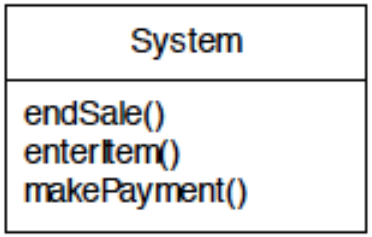
- Low Coupling
  - Jak moc jsem zavislý na mamince
  - Jak jsem samostatný
- High cohesion
  - V práci jsem zodpovědný za programování v C, kafe, management týmu
  - V práci jsem zodpovědný za programování perzistence, znám SQL, DB, JPA, JDBC

Controller

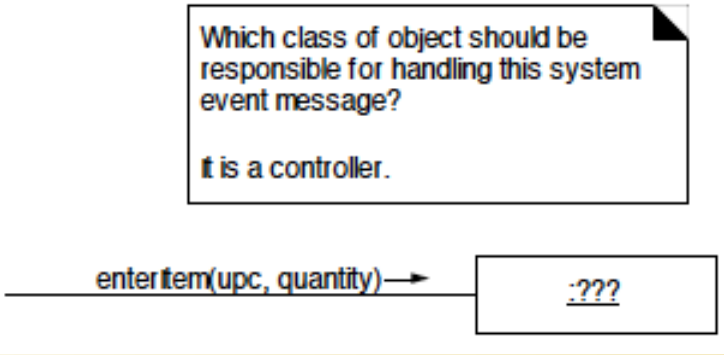
- Problem: Kdo je zodpovědný za řízení vstupních systémových událostí?
- Solution: Přiřaď zodpovědnost k obdržení a zpracování systémové události do třídy reprezentující jedno z následujících:
  - Represents the **overall system** (facade controller).
  - Represents a **use case scenario** (use case handler).
  - Represents from a **real world** (role controller).
  - Controller je ne-user interface objekt, který definuje metodu pro systémovou operaci.
    - Note that windows, applets, etc. typically receive events and delegate them to a controller.

Controller

V POST aplikaci je mnoho systémových operací

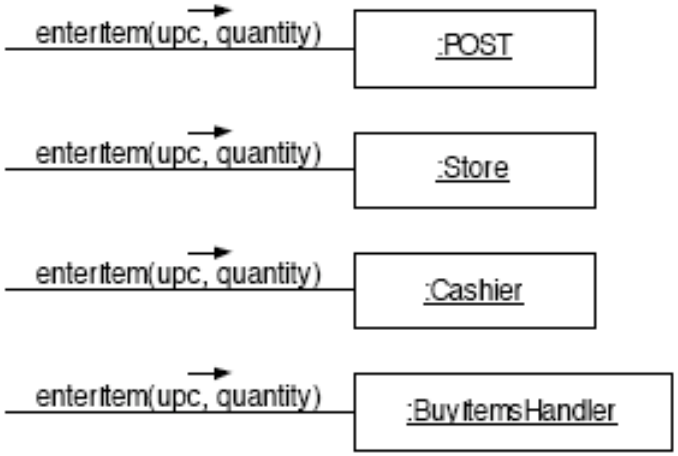


Kdo by za ně měl být zodpovědný?



Controller

- Pomocí Controller pattern, jsou volby:
  - POST–celkový “system”
  - Store–colkový business/organizace
  - Cashier–real-world entita zapojená do úkolů
  - BuyItemsHandler–umělý handler všech systémových operací pro use case

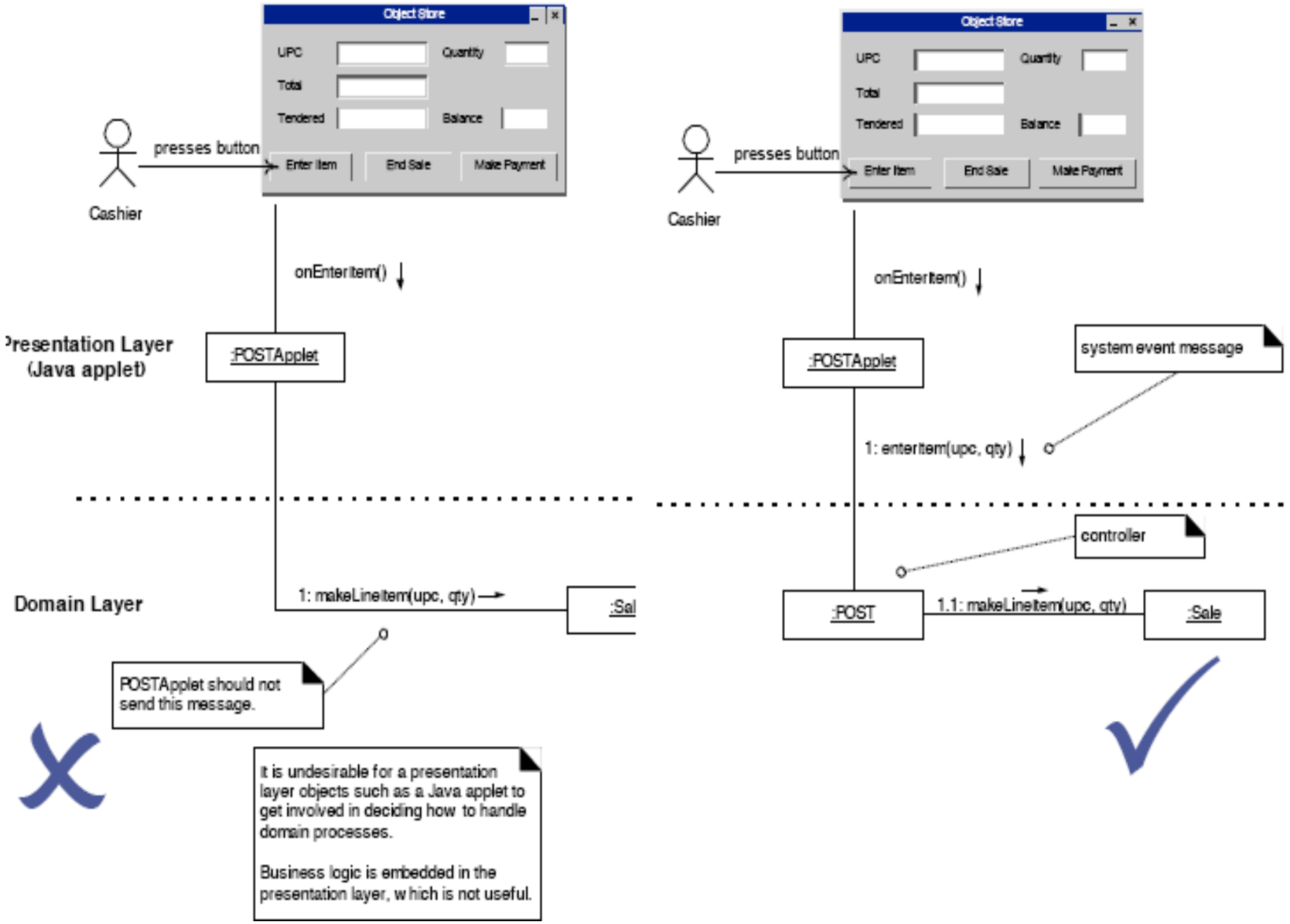


Controller

Správná volba je závislá na dalších faktorech (jako coupling a cohesion) ale doporučení jest:

- Použij **façade** kontroléry pokud jsou se jedná o pár systémových událostí
- **UseCaseHandler** je vhodný pokud máme mnoho systémových událostí mezi různými procesy a façade kotrolery by se staly přeplněné
- Použij **role** kontroléry opatrně.
  - Je snadné přiřadit veškerou práci person-like objektům na místo delegátů

- Note: **Presentation layer does not handle system events**


Object Store
UPC
Quantity
Total
Tendered
Balance
Enter Item
End Sale
Make Payment
presses button
Cashier
onEnterItem()
Presentation Layer
(Java applet)
:POSTApplet
Domain Layer
1: makeLineItem(upc, qty)
:Sal
POSTApplet should not send this message.
It is undesirable for a presentation layer objects such as a Java applet to get involved in deciding how to handle domain processes.
Business logic is embedded in the presentation layer, which is not useful.
Object Store
UPC
Quantity
Total
Tendered
Balance
Enter Item
End Sale
Make Payment
presses button
Cashier
onEnterItem()
:POSTApplet
system event message
1: enterItem(upc, qty)
controller
:POST
1.1: makeLineItem(upc, qty)
:Sale

Controller

Důsledek

- větší potenciál pro reuse = jako MVC
- možné úvahy o stavu use case
  - pokud UseCaseHandler užit pro veškeré systémové události náležící use casu, je možné zajistit systémové operace, tak abych se vyskytly jen v legální sekvenci

Polymorfismus

Záměr

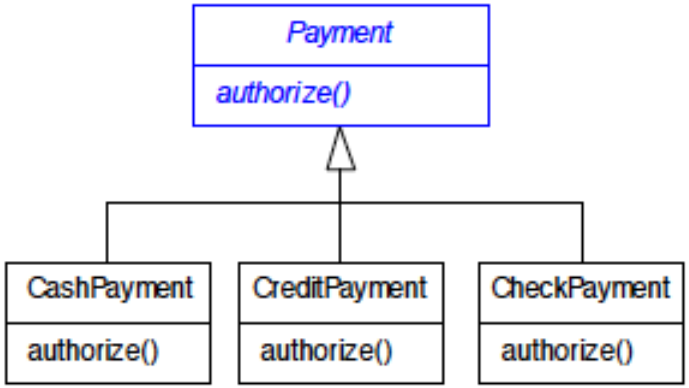
- Pokud se alternativy liší typem, přiřaď zodpovědnosti za chování - pomocí polymorfních operací – typům pro které se chování liší
- Pozor, netestuj typ objektu ale použij podmínku k vybrání alternativy

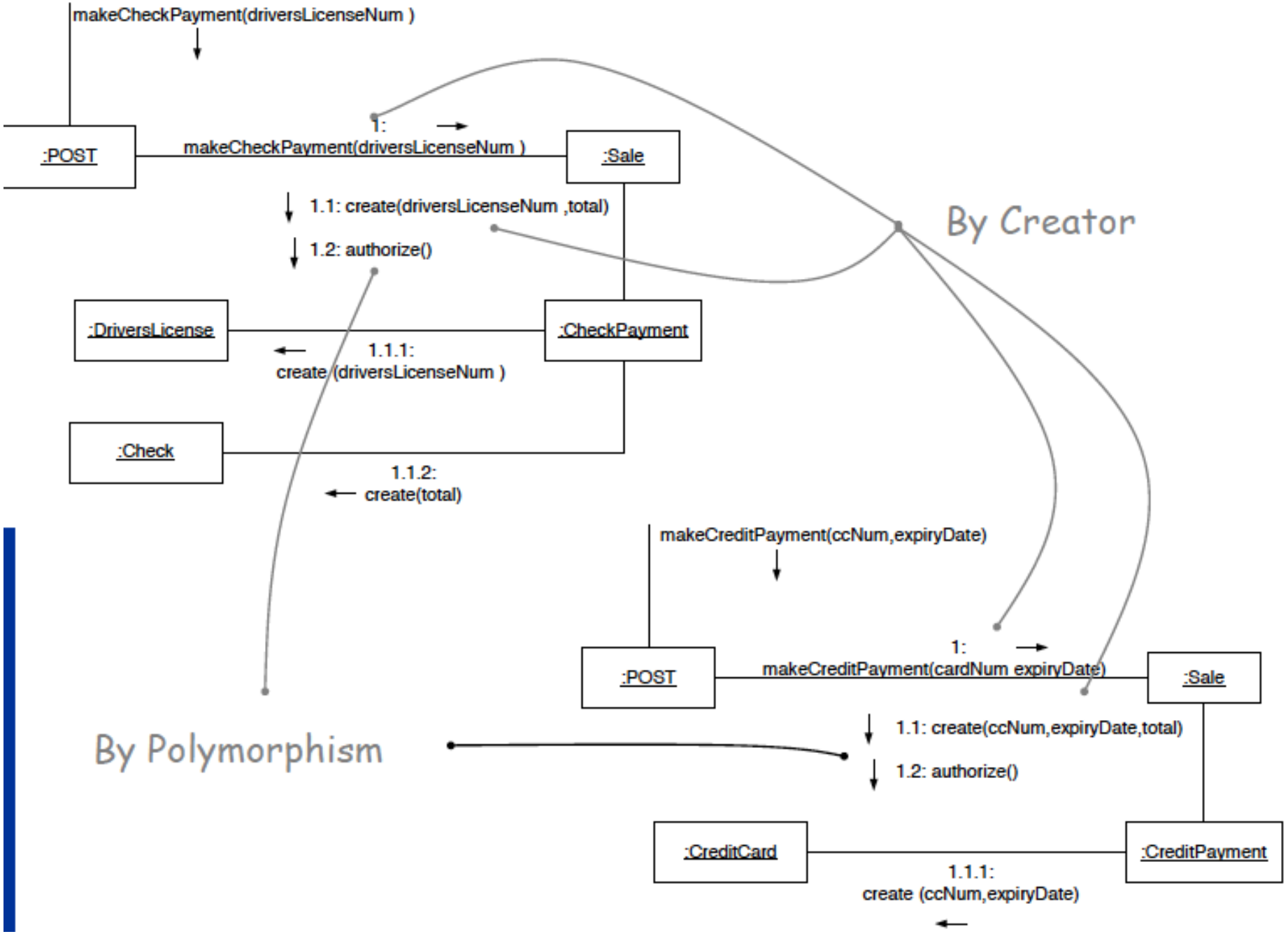
Problém

- Jak řešit alternativy dle typu
    - Pokud program používá podmínky k výběru a nová variace znamená změnu těchto podmínek

Polymorfismus

- V POST aplikaci, kdo je zodpovědný za autorizace různých plateb?
- Akceptovat lze cash, check, credit či debit platby a autorizace je pokaždé různá.
- Z Polymorphismu, přiřadíme zodpovědnost každému typu Payment (inherit interface from abstract *Payment* class)




makeCheckPayment(driversLicenseNum)
:POST
1:
makeCheckPayment(driversLicenseNum)
:Sale
1.1: create(driversLicenseNum ,total)
1.2: authorize()
By Creator
:DriversLicense
:CheckPayment
1.1.1:
create (driversLicenseNum)
:Check
1.1.2:
create(total)
makeCreditPayment(ccNum,expiryDate)
:POST
1:
makeCreditPayment(cardNum expiryDate)
:Sale
1.1: create(ccNum,expiryDate,total)
1.2: authorize()
By Polymorphism
:CreditCard
:CreditPayment
1.1.1:
create (ccNum,expiryDate)

Polymorfismus

Důsledek

- Snadné rozšíření
- Podobné
- Malá závislost

Podobné

- GoF Command, Strategy a State

Pure fabrication

Záměr

- Přiřaď kohezní množinu odpovědností do umělé třídy, která nic nereprezentuje v dané doméně. To potom umožní podporu pro
  - high cohesion, low coupling, and reuse.

- Only do when desperate? We are often desperate!

Problém

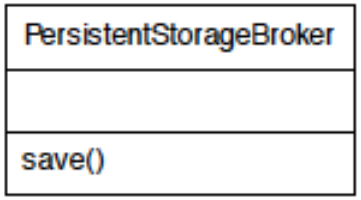
- OO je charakteristické implementací SW tříd jako konceptů reálného světa.
- Co ale pokud přiřazení odpovědností do doménových tříd znaméná špatnou kohezi veliké spárování (tight coupling)?

Pure fabrication

Sale instance jsou uloženy v databázi.
Pomocí Experta by *Sale* měl být za toto zodpovědný.

• Ale:
– Tento úkol požaduje veliké množství operací pro spojení s DB, žádná z nich nemá vztah ke konceptu *Sale* (incohesive)
– *Sale* class musí být spárovaná (coupled) s rozhraním DB
- tedy coupling roste – a to není do jiného doménového objektu!
– Ukládání objektů do DB je obecná úloha kterou využije více tříd. Přiřazením zodpovědnosti do *Sale* znamená špatný reuse a noho duplokace v dalších třídách

• Řešení: vytvoř třídu zodpovědnou jen za ukládání objektů do DB



Pure fabrication

Důsledek

- Reuse, low coupling, high cohesion

Podobné

- GoF Adapter, Visitor, Observer

Indirection

Záměr

- Přiřaď zodpovědnosti do prostředníka aby zprostředkovával komunikaci mezi komponentami servisy, tak aby nebyli přímo spárované

Problém

- Jak odpárovat objekty, tak že low coupling zůstane a reuse je zvýšen

Indirection

V POST aplikaci, terminal musí používat modem k odeslání plateb

.

- OS poskytne low-level API pro přístup k modemu.
- Třída *CreditAuthorizationService* je zodpovědná za komunikaci s modememem.
- ALE, pro propojení nechceme aby detaily modemu byly zaneseny do do doménové třídy – (highly coupled to OS).
- Raději, přidáme prostředníka, třídu *Modem,* která bude reprezentovat rozhraní do doménové třídy. Také známé jako proxy.

Indirection

Důsledek

- Low coupling

Podobné

- GoF: Mediator, Adapter, Facade, Observer

Don’t talk to strangers

Záměr

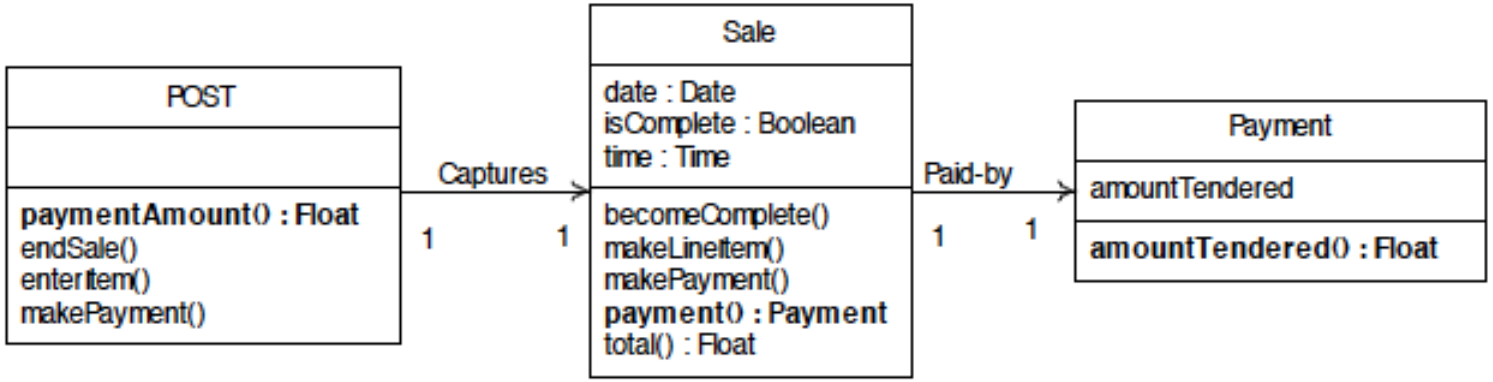
- Přiřaď zodpovědnosti do klientského přímého objektu, ta aby spolupracoval s nepřímím objektem. Tak, že klient neví nic o nepřímém objektu
- Umísťuje omezení na to jaké objekty mohou odesílat zprávy uvnitř metody. Ty by měly být odeslány pouze na:
    - Sebe, či vlastní atribut nebo vlastní kolekci
    - Parametr metody
    - Objekt vytvořen uvnitř metody

Problém

- Pokud objekt má znalosti vnitřní struktury jiného objektu, potom trpí s high coupling.
- Pokud klientský objekt musí použít servis k obdržení informací z nepřímého objektu, jak toho může docílit bez spárování do znalosti vnitřní struktury jeho přímého serveru či nepřímého objektu

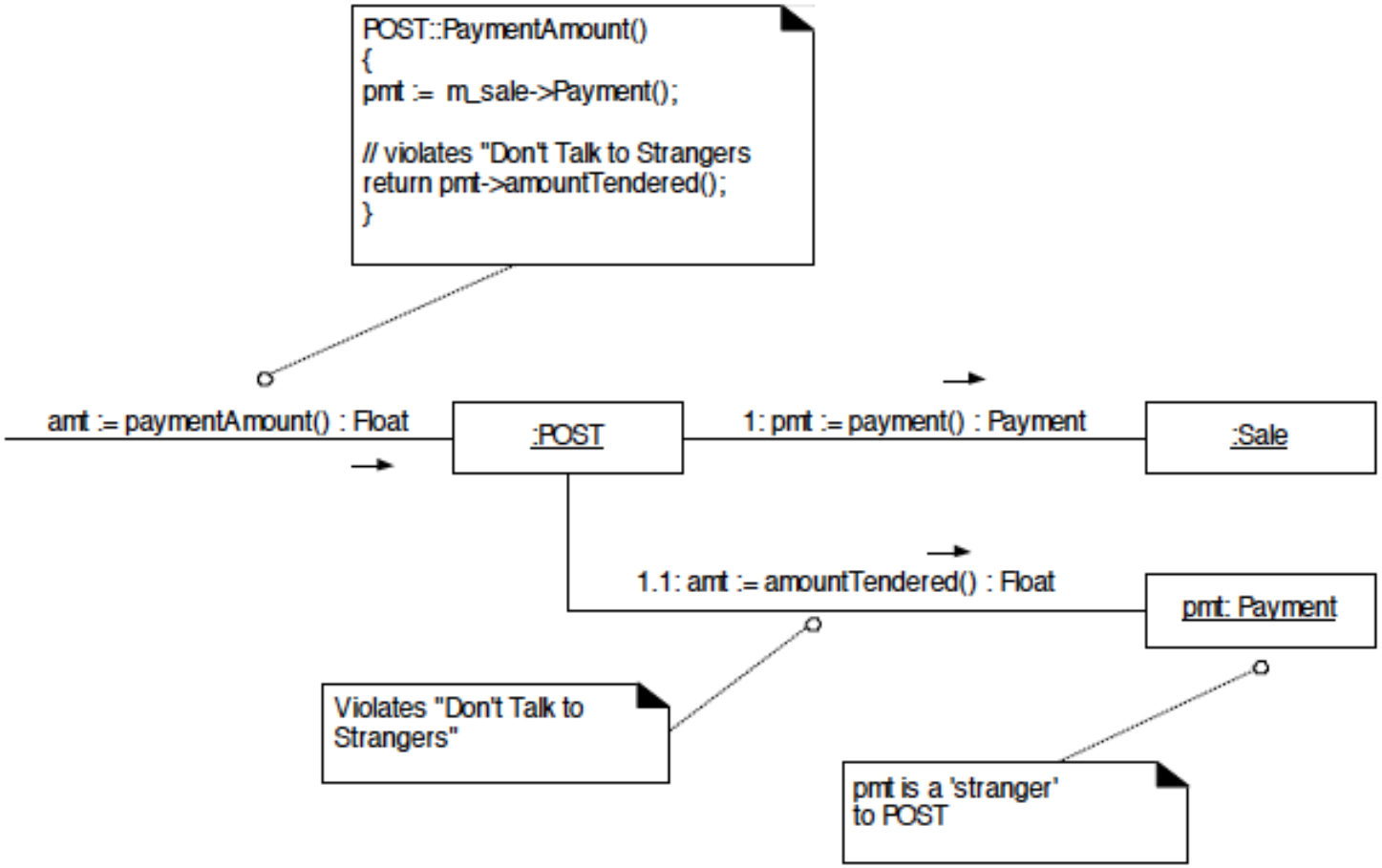
Don’t talk to strangers

- V POST app, *POST* instance obsahuje atribut referující na *Sale*, a ten má atribut referující na *Payment*.
- *POST* instance podporují operaci *paymentAmount*, ta vrací aktualní hodnotu předloženou k platbě.
- *Sale* instance podporují operaci *payment*, ta vrací *Payment* instanci asociovánu se *Sale*.



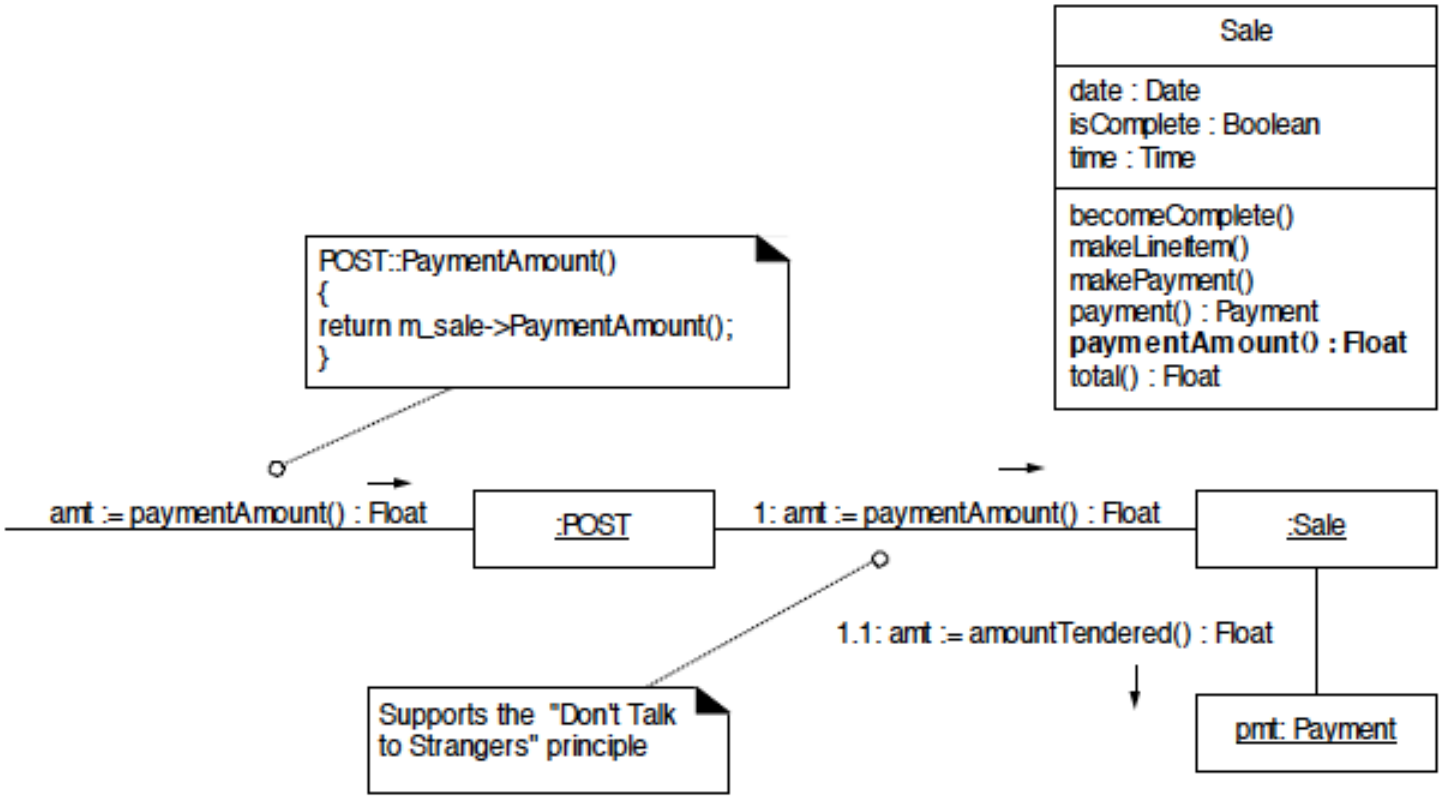
Don’t talk to strangers

Jedna možnost je



Don’t talk to strangers

- Lepší volba je přidat zodpovědnost do přímého objektu (*Sale*), aby vrátil Payment amount do *POST*
- Známé jako **promoting the interface**



Don’t talk to strangers

Proti zákonu

- Jako každý zákon, někdy se musí porušit
- Pokud je “broker” či “object server” zodpovědný za návrat jiných objektů na bázi vyhledání, pak je možné získat viditelnost do těchto objektů skrze broker a zaslat jim zprávu přímo.

Podobné

- Indirection, Chain of Responsibility