The Principles of Object-Oriented Design

Marek Rychlý
rychly@fit.vutbr.cz

Brno University of Technology
Faculty of Information Technology
Department of Information Systems

Information Systems Analysis and Design (AIS)
28 November 2019

Outline

The Principles of OOD

- Were proposed by Robert C. Martin in 1995.
  (a core philosophy for not-only-OO development methodologies)
- Expose the dependency management aspects of OOD.
  (as opposed to the conceptualization and modelling aspects)
- Poor dependency mgmt. leads to code that is difficult maintain.
  (hard to change, fragile, and non-reusable)

1. SOLID principles of class design.
2. RCC principles about package cohesion.
3. ASS principles about couplings between packages and metrics.
   (the metrics of the package structure of a system)

SOLID Principles

(1) The principles of class design (SOLID) are

SRP	The Single Responsibility Principle (a class should have one, and only one, reason to change)
OCP	The Open Closed Principle (you should be able to extend a classes behaviour, without modifying it)
LSP	The Liskov Substitution Principle (derived classes must be substitutable for their base classes)
ISP	The Interface Segregation Principle (make fine grained interfaces that are client specific)
DIP	The Dependency Inversion Principle (depend on abstractions, not on concretions)

RCC and ASS Principles

(2) The principles about package **cohesion** (RCC) are

REP	The Release/Reuse Equivalency Principle (the granule of reuse is the granule of release)
CCP	The Common Closure Principle (classes that change together are packaged together)
CRP	The Common Reuse Principle (classes that are used together are packaged together)

(3) The principles about the **coupling** between packages and metrics of the package structure of a system (ASS) are

ADP	The Acyclic Dependencies Principle (the dependency graph of packages must have no cycles)
SDP	The Stable Dependencies Principle (depend in the direction of stability)
SAP	The Stable Abstractions Principle (abstractness increases with stability)

SOLID: The Single Responsibility Principle (SRP)

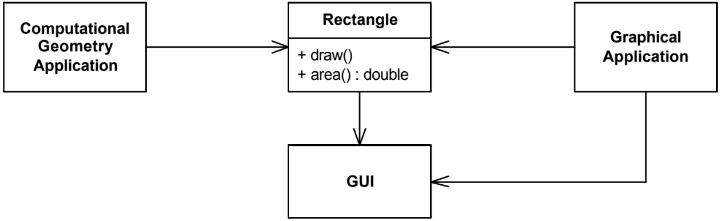
"a class should have one, and only one, reason to change"

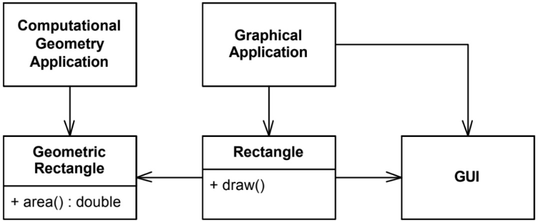
- Every class should implement a single part of the functionality.
  (it need not to be one operation, but several operations with a **high cohesion**[1])
- It should take responsibility over this functionality only.
  (and this responsibility should be entirely encapsulated by the class)
- Each responsibility is "a reason for change / an axis of change"[2].
  (changing requirements result into a change in responsibility of related classes)
  - Responsibilities in a class become coupled.
    (changes to one responsibility may impair or inhibit the ability of the class to meet the other its responsibilities)
  - Separate unrelated responsibilities into separate classes.
    (a "god" class with many different operations should be refactored)
  - However, if two responsibilities change always at the same times, then there is no need to separate them (it is a high cohesion).
    (an axis of change is only an axis of change if the changes actually occur)

[1] Cohesion is the functional relatedness of the elements of a module/class.
[2] More than one motive for changing a class $\rightarrow$ class has more than one responsibility.

An Example of Shared/Separated Responsibilities

(if possible, separate implementation into different classes)

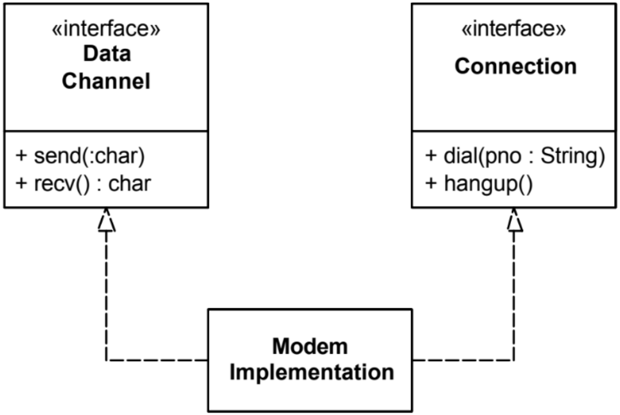




(adopted from "Agile Software Development: Principles, Patterns, and Practices" by R. C. Martin)

An Example of Separated Resps. by Interfaces

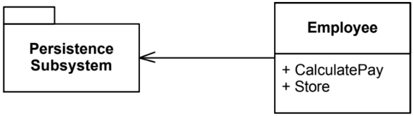
(if the implementation cannot be separated, separate declarations into different interfaces)



(adopted from “Agile Software Development: Principles, Patterns, and Practices” by R. C. Martin)

Persistence as a Responsibility

- Do not mix business rules and persistence control responsibilities.
- They usually change at different times.
  (and with various frequency; business rules change more frequently)
- They always changes for completely different reasons.
- Use the Facade, DAO, or Proxy patterns to separate them.



(adopted from “Agile Software Development: Principles, Patterns, and Practices” by Robert C. Martin)

SOLID: The Open Closed Principle (OCP)

“you should be able to extend a classes behaviour, without modifying it”

- Classes/modules should be
  - “open for extension”
    (this means that their behaviour can be extended as required)
  - and “closed for modification”.
    (their source code is inviolate; they are stable and available for use by others)
- Design classes that never change (except for bug-fixes).
  (extend the behaviour of such classes by adding new code, e.g., in subclasses, not by changing old code that already works)
- A change of a class should not propagate in a cascade.
  (a single change to a class should not result in a cascade of changes to dependent classes that can break a system)
- Can be achieved by abstraction (interfaces, abstract operations).
  - define a stable interface and use them instead of full classes,
  - use visibility modifiers for attributes/methods,
  - think about future changes and make them possible.
    (the extension points; e.g., by design patterns)

SOLID: The Liskov Substitution Principle (LSP)

“derived classes must be substitutable for their base classes”

Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T. — *Liskov/Wing (1994)*

- Two similarly looking objects may not be related via specialisation.
  (unless they behave according to the LSP; e.g., a square is not a rectangle)
- Object specialisation is a (strong) behavioural sub-typing.
  (“is a” relationship pertains to extrinsic public behaviour that clients depend upon)
- According to “Design by Contract” of Bertrand Meyer
  (where behaviour can be defined by a contract, i.e., by pre- and postconditions)
  - overridden methods must have the same or weaker preconditions,
  - and the same or stronger postconditions.
- LSP can be broken by object reflection/inspection.
  (the encapsulation must respected; a class membership must be check by “is a” predicate, not by any other means)

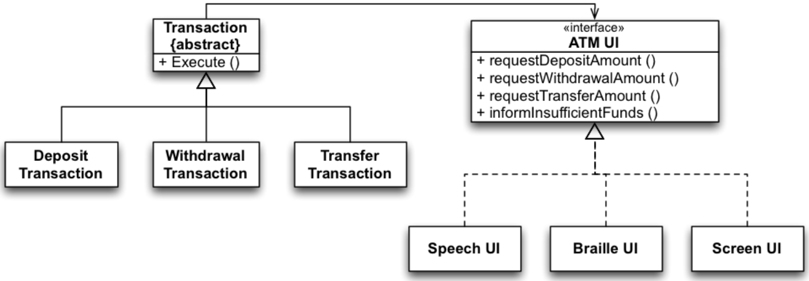
SOLID: The Interface Segregation Principle (ISP)

"make fine grained interfaces that are client specific"

- Do not force a client to depend upon an interface it does not use.
  (otherwise, the client is subject to changes to the interface, e.g., the changes that other clients utilising the interface force upon it)
- Also known as "the syndrome of interface pollution":
  A base class is forced to incorporate/use a new interface of other class solely for the benefit of one of its subclasses that needs the incorporated.
- Broke up polluted/fat interfaces into groups of member functions, each high-cohesion group serves a different set of clients.
  (classes that have "fat" interfaces are classes whose interfaces are not cohesive)
- This works well with the Single Responsibility Principle (SRP).
  (see the SRP example on page 8, responsibilities were separated by interfaces)

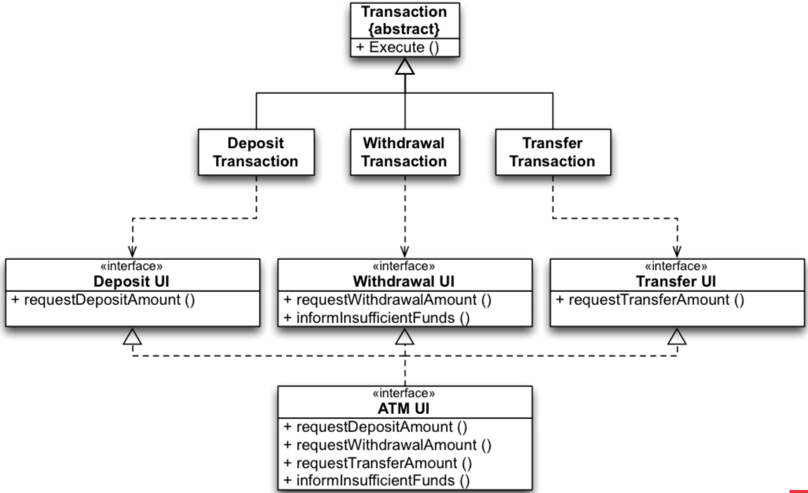
An Example of a Polluted Interface

- ATM is required to support transactions: withdraw, deposit, and transfer.
- There must be support for different languages and different kinds of UIs.
- Each transaction class needs to call methods on the GUI.
  (e.g., to ask for the amount to deposit, withdraw, transfer)



(adopted from “Software Engineering Design & Construction” by Michael Eichberg)

The Example with ISP Compliant Interfaces


Transaction {abstract}
+ Execute ()
Deposit Transaction
Withdrawal Transaction
Transfer Transaction
«interface» Deposit UI
+ requestDepositAmount ()
«interface» Withdrawal UI
+ requestWithdrawalAmount ()
+ informInsufficientFunds ()
«interface» Transfer UI
+ requestTransferAmount ()
«interface» ATM UI
+ requestDepositAmount ()
+ requestWithdrawalAmount ()
+ requestTransferAmount ()
+ informInsufficientFunds ()

(adopted from “Software Engineering Design & Construction” by Michael Eichberg)

SOLID: The Dependency Inversion Principle (DIP)

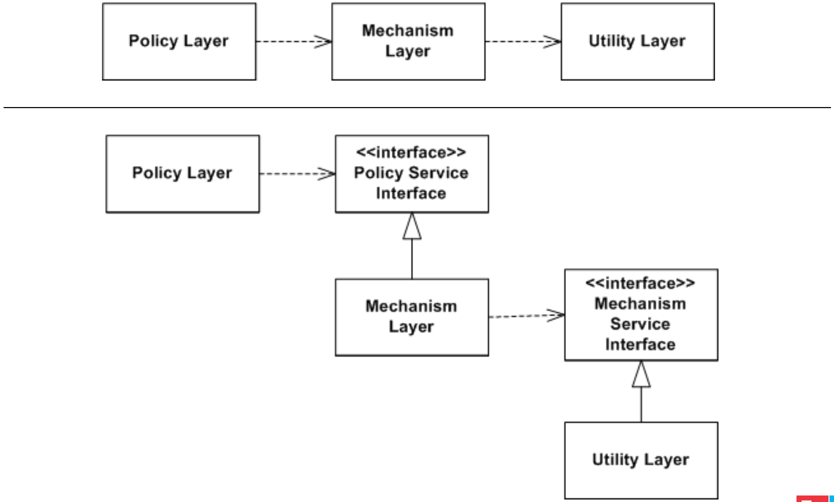
"depend on abstractions, not on concretions" (according to Robert C. Martin)

- Dependencies complicate software design → bad design:

Rigidity	It is hard to change because every change affects too many other parts of the system.
Fragility	When you make a change, unexpected parts of the system break.
Immobility	It is hard to reuse in another application because it cannot be disentangled from the current application.

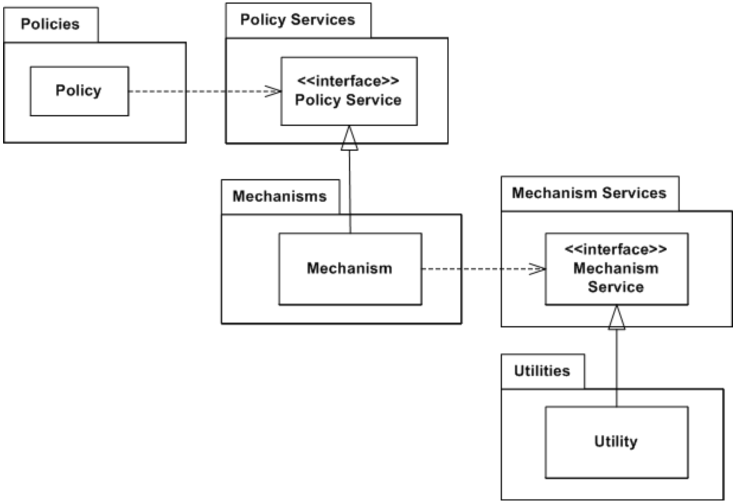
- The Dependency Inversion Principle says that
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.
- We need to introduce those abstractions – e.g., service interfaces.

An Example of Dependencies without and with DIP



(adopted from “Dependency inversion principle” by Wikipedia)

The Example with Packages



(adopted from “Dependency inversion principle” by Wikipedia)

REC: The Release/Reuse Equiv. Principle (REP)

“the granule of reuse is the granule of release”

- Class is too finely grained to be used as an organizational unit.
  (we need to group the classes into packages)
- Package is a granule/unit of reuse.
  (classes that are used together are packaged together, see page CRP on 21)
- . . . and it is also a granule/unit of release.
  (classes that change together are packaged together, see CCP on page 20)
- Source code should not be reused by copy&paste.
  (copying it from one class and pasting it into another)
- The code should be reused by including a released library/package with a stable API (of a particular version).
  (the library/package needs to be versioned, to track modifications)

REC: The Common Closure Principle (CCP)

“classes that change together are packaged together”

- It is necessary to keep the high cohesion inside a package.
  (maintainability is more important than re-usability)
- The classes in a package should be closed together against the same kinds of changes.
  (such classes are tightly bound, either physically or conceptually)
- A change that affects a package affects all its classes.
- However, 100% closure is not attainable as the classes should be closed for modification but open for extension.
  (see the Open Closed Principle/OCP on page 11)
- Some dependent packages may be also affected by the change.
  (yet, as CCP is grouping similar classes into the same packages, the change has a good chance of being restricted to a minimal number of packages)

REC: The Common Reuse Principle (CRP)

"classes that are used together are packaged together"

- It helps to decide which classes should be placed into a package.
- The classes in a package are reused together.
  (they are part of the reusable abstraction, a package)
- If you reuse one of the classes in a package, you reuse them all.
  (the classes collaborate with other classes in the package)
- A dependence on a package $\rightarrow$ a dependence on all its classes.
  (otherwise, there would be useless classes and their revalidating on integration and redistributing on deployment would be a waste of effort[3])

[3]on each release of a used package, a using pkg. must be revalidated/re-released

ASS: The Acyclic Dependencies Principle (ADP)

“the dependency graph of packages must have no cycles”

- Package is a unit of work, a responsibility of a development team.
  (there are two versions of packages: development and stable/released)
- When the team gets a package working, it is released for use by the other teams (they can use the released version only).
  (the package get a release number and is published/deployed into a repository)
- Other teams need not to immediately adopt the new release.
  (they can simply continue using the old release and switch when ready)
- However, to make it work there can be no cyclic dependencies.
  (that is a package dependency graph is a directed acyclic graph)
- The cycles can be broken
  - by an application of the Dependency Inversion Principle/DIP, (see page 16)
  - or by an extraction of problematic classes into a new package. (see also one of the previous lectures)

ASS: The Stable Dependencies Principle (SDP)

"depend in the direction of stability"

- The dependencies between packages in a design should be in the direction of the stability of the packages.
  (a package should only depend upon packages that are more stable than it is)
- Targets of dependencies should be extremely unlikely to change.
  (that is stable, non-volatile, with releases not breaking their API/ABI)
- Stability is a measure of the difficulty in changing a module.
  (it is not only a measure of the likelihood that a module will change)
- Modules more difficult to change, are going to be less volatile.
- The stability can be achieved
  - by independent classes,
    (the classes that do not depend upon anything else; so a change from a dependee cannot ripple up to them and cause them to change)
  - and by responsible classes.
    (the classes that tend to be stable because any change has a large impact; the more dependents they have, the harder it is to make changes to them)

Package Stability Metrics
(according to Robert C. Martin)

- Afferent Couplings: C_a
  (a number of classes outside a package that depend upon classes within the pkg.)
- Efferent Couplings: C_e
  (a number of classes inside a package that depend upon classes outside the pkg.)
- Positional Instability: $I = \frac{C_e}{C_a + C_e}$
  (in range $[0, 1]$ for a maximally stable to a maximally unstable package)
- For $I = 1$, a package is very unstable, irresponsible & dependent.
  (there are no other packages depended upon the package and this package does depend upon other packages)
- For $I = 0$, a package is very stable, responsible & independent.
  (the package is depended upon by other packages, but does not itself depend upon any other packages)
- Packages in the highest architectural layers need not to be stable.
  (they are depended upon packages in lower layers, but there are no other packages depended upon them)

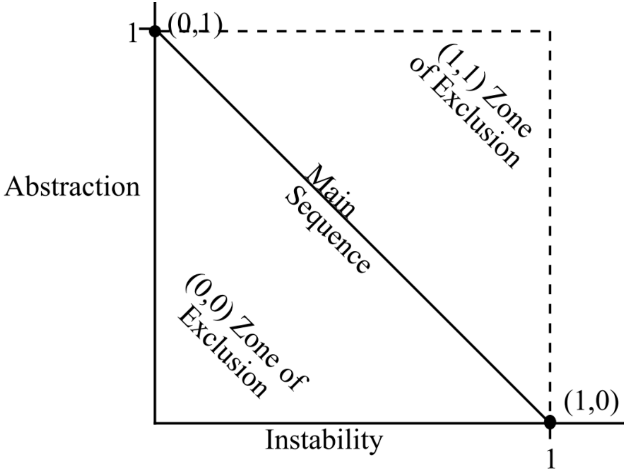
ASS: The Stable Abstractions Principle (SAP)

“abstractness increases with stability”

- Packages that are maximally stable should be maximally abstract and unstable packages should be concrete.
  (i.e., the abstraction of a package should be in proportion to its stability)
  - The stability does not prevent abstract pkgs. from being extended.
  - The instability allows a concrete code in a pkg. to be easily changed.
- Each stable package should be flexible and extensible.
  (should not constrain the design; be flexible and extensible, e.g., by publishing its interfaces, abstract classes, etc.)
- Abstraction Metric: $A = \frac{\text{number of interfaces and abstract classes}}{\text{total number of interfaces and classes}}$
  (in range [0, 1] for a package with no abstract classes at all to a package with nothing but abstract classes)

The Abstraction/Instability Relationship

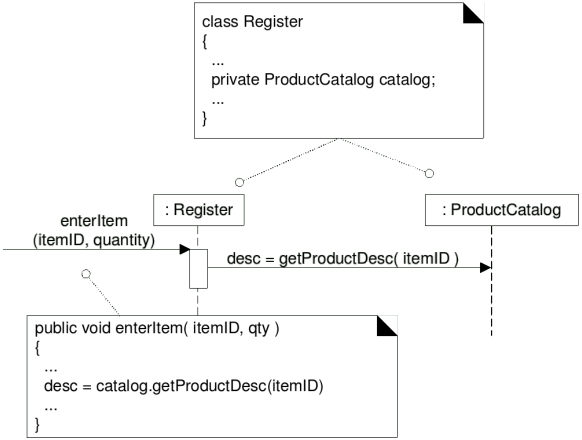
(it usually does not make sense to have a package in the zone of exclusion: pain/usefulness)



(adopted from “Stability” by Robert C. Martin)

Visibility of Objects

The ability of an object to see another object by its reference.
(e.g., “enterItem” method below use privately visible attr. with “ProductCatalogue” object)



```
class Register
{
  ...
  private ProductCatalog catalog;
  ...
}
```

```
public void enterItem( itemID, qty )
{
  ...
  desc = catalog.getProductDesc(itemID)
  ...
}
```

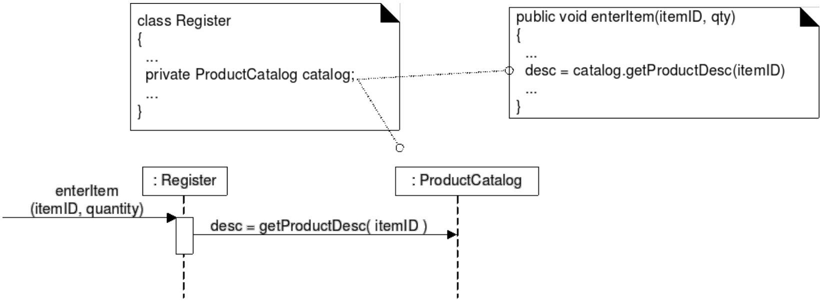
(adopted from “Applying UML and Patterns” by Craig Larman)

Ways to Achieve the Object Visibility

- **Attribute visibility**
  (*B* is an attribute of obj. *A*)
- **Parameter visibility**
  (*B* is a parameter of an operation/method in obj. *A*)
- **Local visibility**
  (*B* is a local object/variable in an operation/method of obj. *A*)
- **Global visibility**
  (*B* is in a global scope, e.g., in a global variable, a runtime context, etc.)

Attribute Visibility

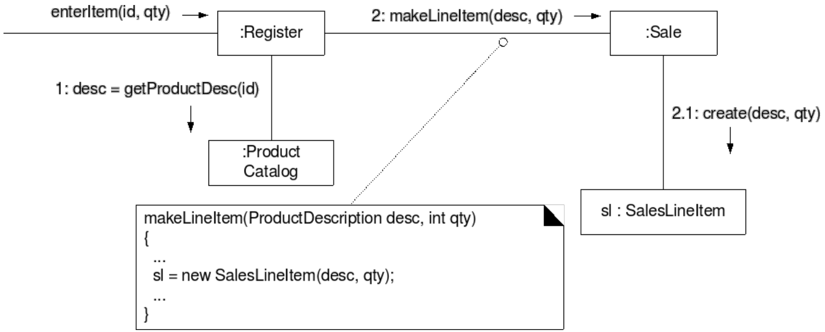
To see another object by its reference in an attribute.
(e.g., “enterItem” method below use privately visible attr. with “ProductCatalogue” object)


class Register
{
...
private ProductCatalog catalog;
...
}
public void enterItem(itemID, qty)
{
...
desc = catalog.getProductDesc(itemID)
...
}
: Register
: ProductCatalog
enterItem
(itemID, quantity)
desc = getProductDesc(itemID)

(adopted from “Applying UML and Patterns” by Craig Larman)

Parameter Visibility

To see another object by its reference in a method's parameter.
(e.g., "makeLineItem" method below use its parameter with "ProductDescription" object)

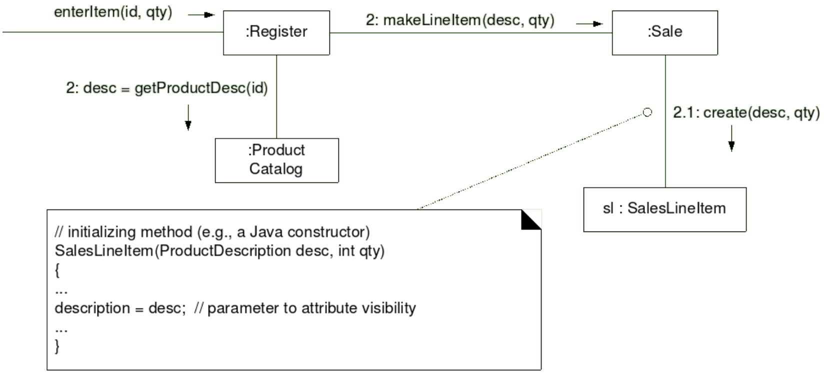


```
makeLineItem(ProductDescription desc, int qty)
{
  ...
  sl = new SalesLineItem(desc, qty);
  ...
}
```

(adopted from "Applying UML and Patterns" by Craig Larman)

Parameter To Attribute Visibility

To assign an object from a method's parameter into an attribute.
(e.g., "SalesLineItem" constructor below use its parameter with "ProductDescription" object to set its attribute; i.e., it is passing the visibility)

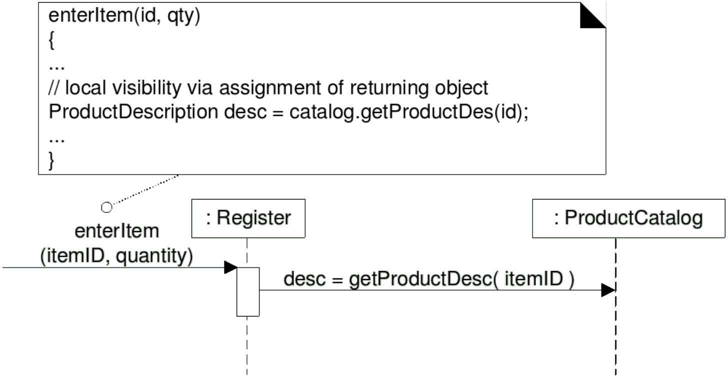


```
// initializing method (e.g., a Java constructor)
SalesLineItem(ProductDescription desc, int qty)
{
...
description = desc;  // parameter to attribute visibility
...
}
```

(adopted from "Applying UML and Patterns" by Craig Larman)

Local Visibility

To assign a new or received object to a local variable.
(e.g., “enterItem” method below receives a “ProductDescription” object by “getProductDesc” method call and assign the object to a local variable)


```
enterItem(id, qty)
{
...
// local visibility via assignment of returning object
ProductDescription desc = catalog.getProductDes(id);
...
}
```

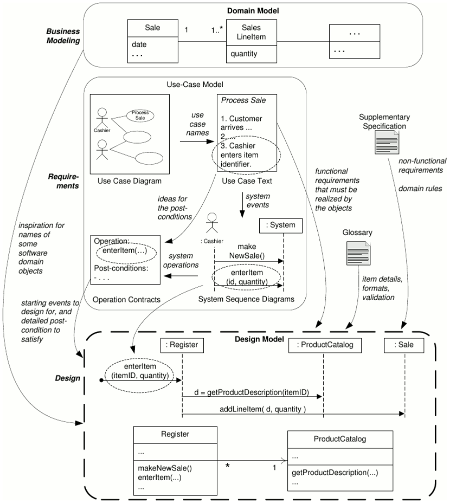
(adopted from “Applying UML and Patterns” by Craig Larman)

Global Visibility

To utilise an object available in a global context.

- This can be achieved by accessing a global variable,
- or by invoking an operation on a singleton object.
  (in implementation frameworks, such singletons are often provided by a runtime environment, e.g., by application containers)

Design Models and Domain/Use Case Models

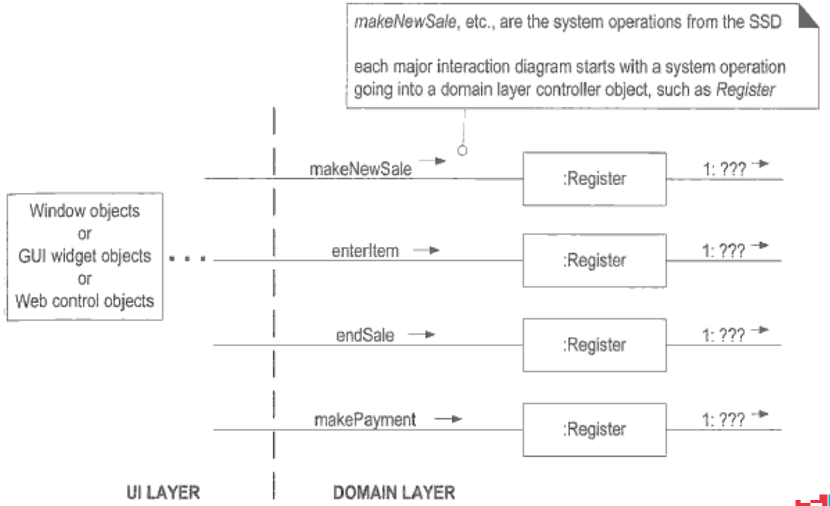


(adopted from “Applying UML and Patterns” by Craig Larman)

NextGen POS: Use Cases for 1st Elaboration Iteration

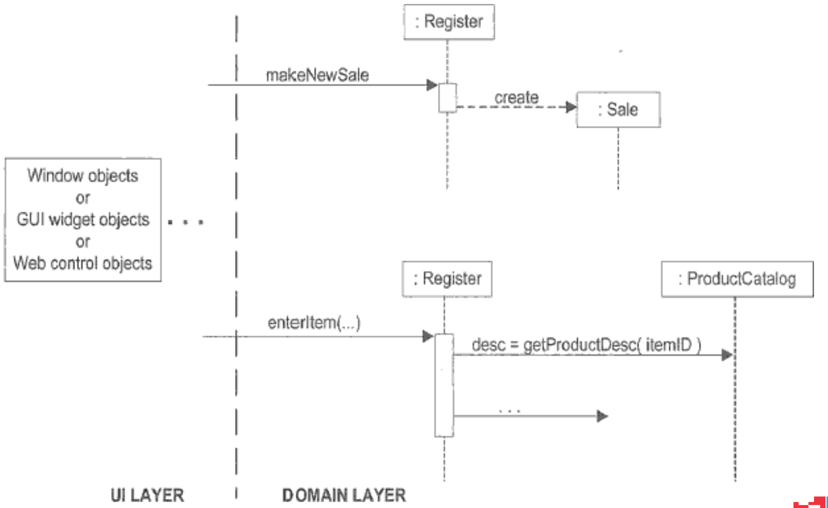
- Process Sale scenario with the following system operations:
  - makeNewSale
  - enterItem
  - endSale
  - makePayment
- Start Up scenario with the following system operation:
  - create (or startUp)

The System Operations in SSD (1)


makeNewSale, etc., are the system operations from the SSD
each major interaction diagram starts with a system operation going into a domain layer controller object, such as *Register*
makeNewSale
:Register
1: ???
Window objects or GUI widget objects or Web control objects
enterItem
:Register
1: ???
endSale
:Register
1: ???
makePayment
:Register
1: ???
UI LAYER
DOMAIN LAYER

(adopted from "Applying UML and Patterns" by Craig Larman)

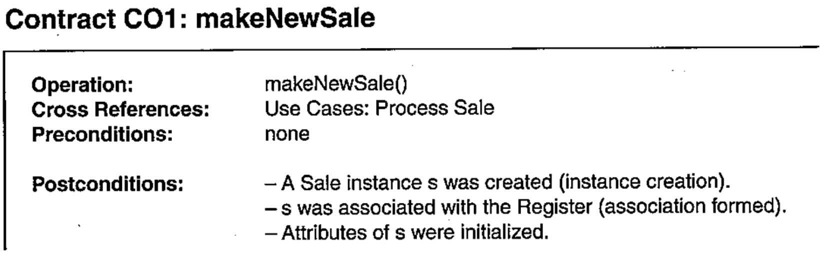
The System Operations in SSD (2)



(adopted from “Applying UML and Patterns” by Craig Larman)

Process Sale: makeNewSale Operation Contract

The postconditions indicate responsibilities to assign.



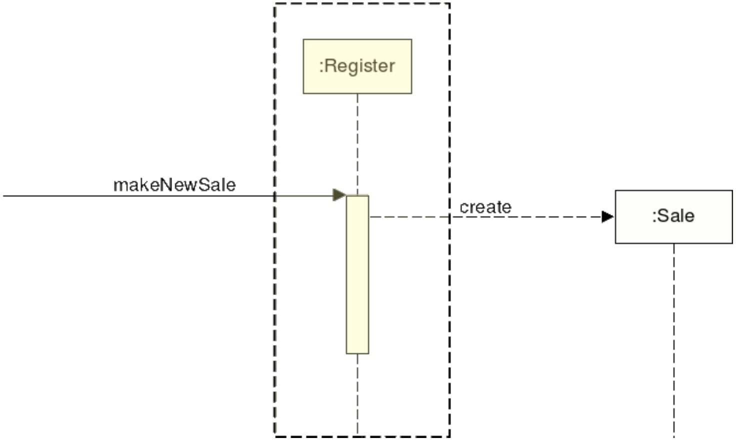
Contract CO1: makeNewSale

Operation:	makeNewSale()
Cross References:	Use Cases: Process Sale
Preconditions:	none
Postconditions:	– A Sale instance s was created (instance creation). – s was associated with the Register (association formed). – Attributes of s were initialized.

(adopted from “Applying UML and Patterns” by Craig Larman)

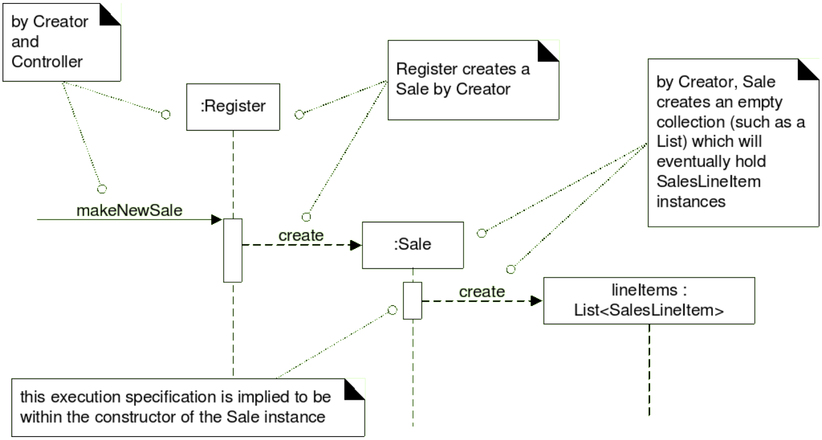
Assign the Controller Role (GRASP)

There is just a few system operations → one controller will be enough.



(adopted from “Applying UML and Patterns” by Craig Larman)

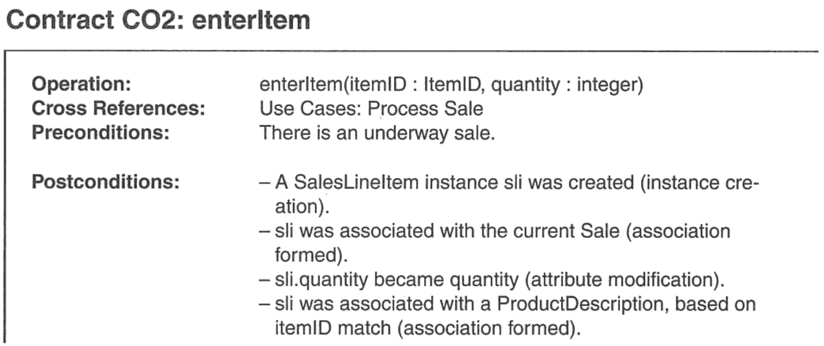
A Creator of Sale Instances (GRASP)


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

(adopted from “Applying UML and Patterns” by Craig Larman)

Process Sale: enterItem Operation Contract

The postconditions indicate responsibilities to assign.



Contract CO2: enterItem

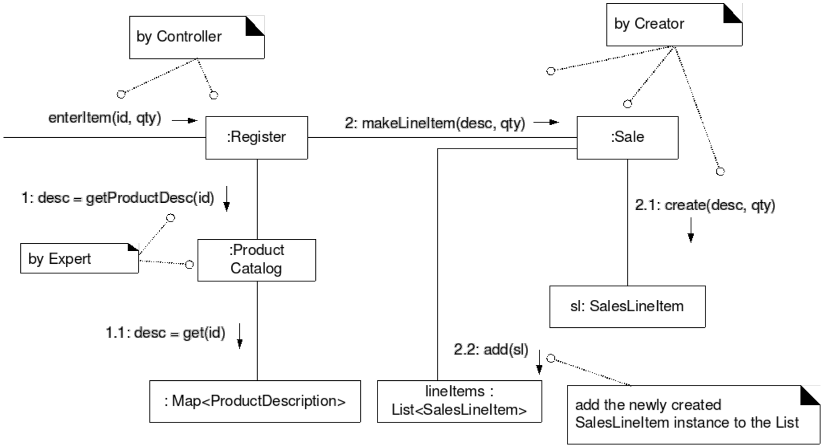
Operation:	enterItem(itemID : ItemID, quantity : integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	– A SalesLineItem instance sli was created (instance creation). – sli was associated with the current Sale (association formed). – sli.quantity became quantity (attribute modification). – sli was associated with a ProductDescription, based on itemID match (association formed).

(adopted from “Applying UML and Patterns” by Craig Larman)

The Controller and a Product's Description & Price

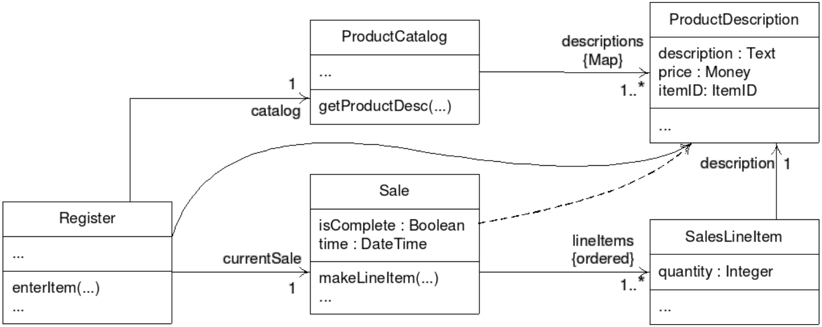
- The controller will be the Register class (see page 41).
- According to the Model-View Separation principle, a product's description and price should not be a responsibility of objects in a view layer, but in the model layer. The view layer's object should be responsible for getting this information from the model layer.
- SaleLineItem objects should be created by a Sale object.
  (they are associated with this object in the domain model and it is responsible according to the Creator principle in GRASP)
- To get the product's information, we need an association from SaleLineItem to ProductDescription objects.
  - ProductCatalog will be responsible for this information.
    (it is an information expert and knows ProductDescription objects)
  - Register will send getProductDescription to the catalogue.
    (it is an information expert and knows the ProductCatalog object)

getProductDescription in a Communication Diagram


by Controller
by Creator
enterItem(id, qty)
:Register
2: makeLineItem(desc, qty)
:Sale
1: desc = getProductDesc(id)
2.1: create(desc, qty)
by Expert
:Product Catalog
sl: SalesLineItem
1.1: desc = get(id)
2.2: add(sl)
: Map<ProductDescription>
lineItems : List<SalesLineItem>
add the newly created SalesLineItem instance to the List

(adopted from “Applying UML and Patterns” by Craig Larman)

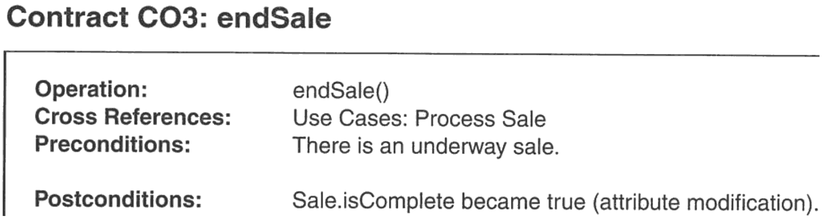
Classes in the Design Model (So Far)



(adopted from “Applying UML and Patterns” by Craig Larman)

Process Sale: endSale Operation Contract

The postconditions indicate responsibilities to assign.
(the controller will be the Register class, see page 41)



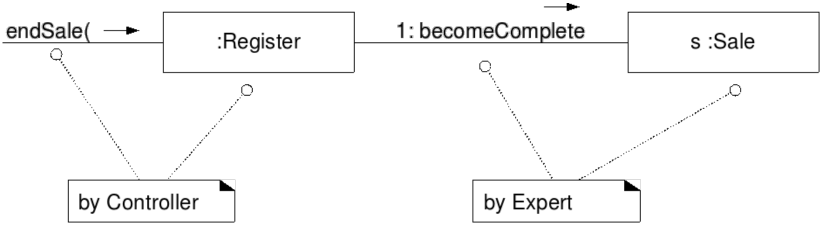
Contract CO3: endSale

Operation:	endSale()
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	Sale.isComplete became true (attribute modification).

(adopted from “Applying UML and Patterns” by Craig Larman)

Setting Sale.isComplete Attribute

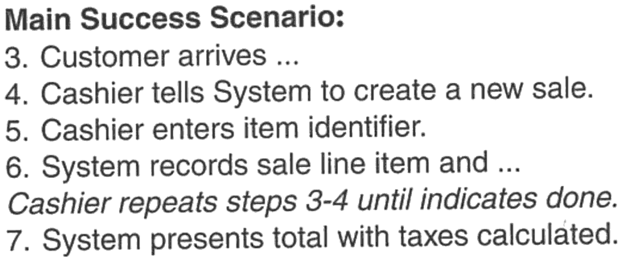
Sale will be responsible, it is an information expert.
(it owns the attribute)



(adopted from “Applying UML and Patterns” by Craig Larman)

Process Sale: getTotal Operation Contract

- It is required by the 7th step of the Process Sale scenario.
- We are at the model layer, not the presentation layer, so we need to assign a responsibility for the information to an information expert domain class.
  (it can be computed from all items as $\sum$ quantity $\times$ unit price)



Main Success Scenario:

3. Customer arrives ...
4. Cashier tells System to create a new sale.
5. Cashier enters item identifier.
6. System records sale line item and ...

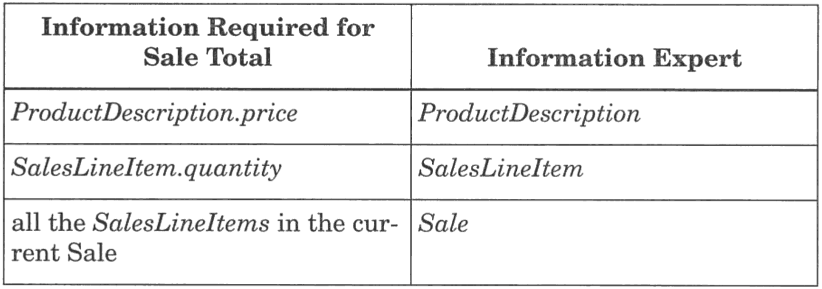
Cashier repeats steps 3-4 until indicates done.

7. System presents total with taxes calculated.

(adopted from “Applying UML and Patterns” by Craig Larman)

Responsibilities for getTotal Operation

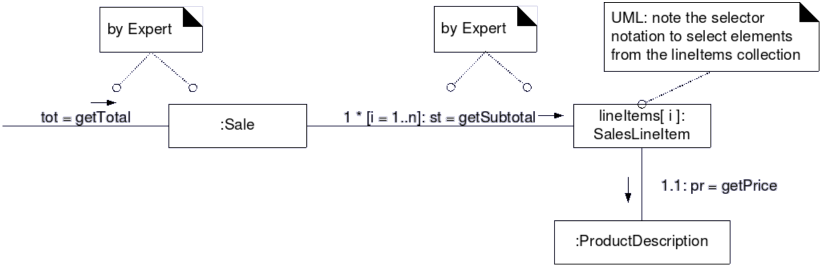
- The total will be computed by the Sale object.
- The price for each line will be computed by SaleLineItem.
- The unit price will be provided by ProductDescription.



Information Required for Sale Total	**Information Expert**
ProductDescription.price	*ProductDescription*
SalesLineItem.quantity	*SalesLineItem*
all the *SalesLineItems* in the current Sale	*Sale*

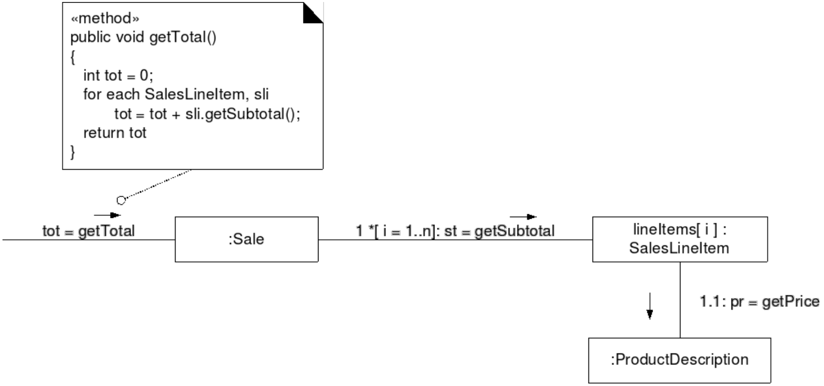
(adopted from “Applying UML and Patterns” by Craig Larman)

getTotal in a Communication Diagram (1)


by Expert
by Expert
UML: note the selector notation to select elements from the lineItems collection
tot = getTotal
:Sale
1 * [i = 1..n]: st = getSubtotal
lineItems[i]:
SalesLineItem
1.1: pr = getPrice
:ProductDescription

(adopted from “Applying UML and Patterns” by Craig Larman)

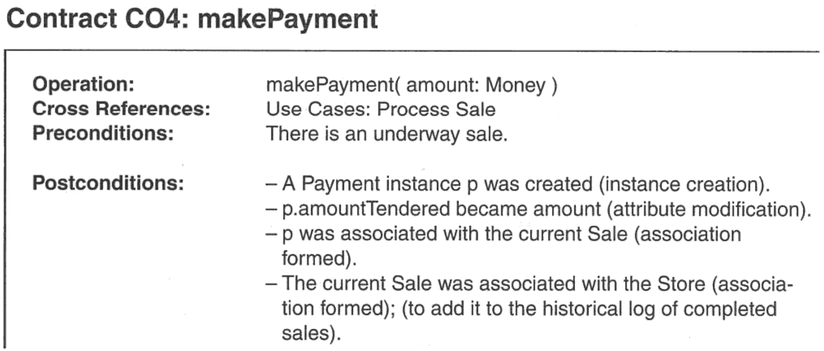
getTotal in a Communication Diagram (2)


```
«method»
public void getTotal()
{
  int tot = 0;
  for each SalesLineItem, sli
        tot = tot + sli.getSubtotal();
  return tot
}
```

(adopted from “Applying UML and Patterns” by Craig Larman)

Process Sale: makePayment Operation Contract

The postconditions indicate responsibilities to assign.



Contract CO4: makePayment

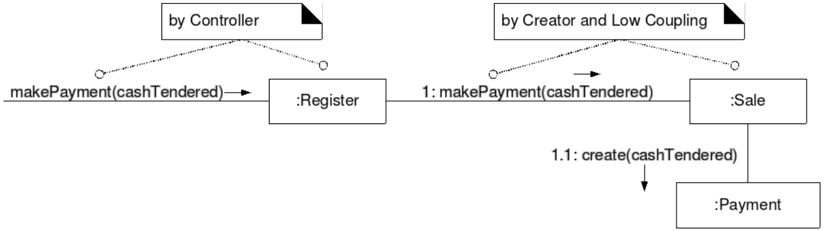
Operation:	makePayment(amount: Money)
Cross References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	– A Payment instance p was created (instance creation). – p.amountTendered became amount (attribute modification). – p was associated with the current Sale (association formed). – The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales).

(adopted from “Applying UML and Patterns” by Craig Larman)

The Controller and a Creator for Payment Instances

- The controller will be the Register class (see page 41).
- Who will create Payment instances?
  - Register – it accepts payment in the real world.
    (knows how the payment was done, amount, etc.)
  - Sale – it is tightly coupled with the Payment class.
    (high cohesion, better maintainability in a future development)
- Sale class is the right choice for the creator.
  (it will help and create payments for the Register object)

makePayment in a Communication Diagram


by Controller
by Creator and Low Coupling
makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
:Sale
1.1: create(cashTendered)
:Payment

(adopted from “Applying UML and Patterns” by Craig Larman)

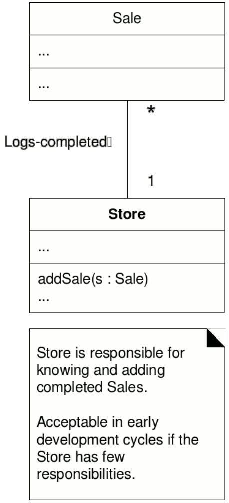
The postconditions are met by this communication.

- A Payment instance was created.
- The instance was associated with a Sale object.
- A cash amount is set in the Payment instance.

A Historical Log of Completed Sales

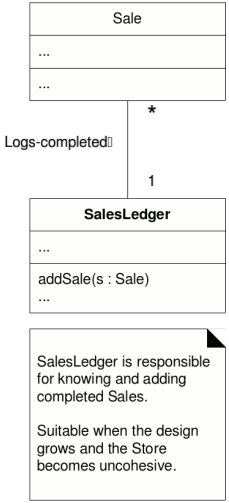
- The Sales Log is one of the user requirements.
  (see the last postcondition of makePayment on page 53)
- This responsibility can be assigned to
  - Store – it is a domain object where the sales happen,
  - SalesLedger – it is necessary for financial accounting.
- Store class is the right choice for the postcondition.
  (however, it is not the only choice and later, the situation can be different; e.g., when the design grows and the Store becomes incohesive)

Responsible Assignments for addSale Operation



Store is responsible for knowing and adding completed Sales.

Acceptable in early development cycles if the Store has few responsibilities.

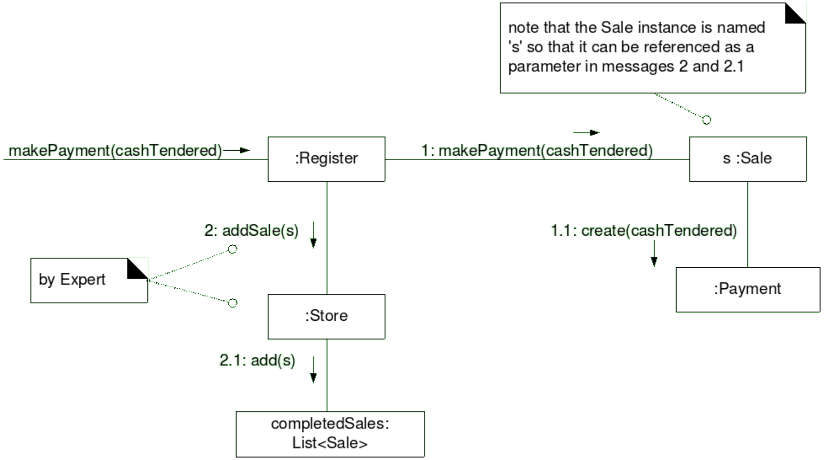


SalesLedger is responsible for knowing and adding completed Sales.

Suitable when the design grows and the Store becomes uncohesive.

(adopted from “Applying UML and Patterns” by Craig Larman)

addSale Operation in a Communication Diagram

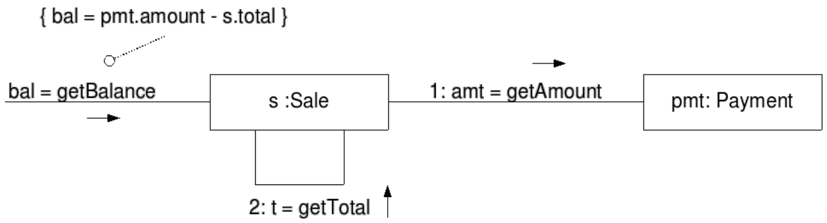

note that the Sale instance is named 's' so that it can be referenced as a parameter in messages 2 and 2.1
makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
s :Sale
2: addSale(s)
1.1: create(cashTendered)
by Expert
:Store
:Payment
2.1: add(s)
completedSales: List<Sale>

(adopted from “Applying UML and Patterns” by Craig Larman)

Getting a Balance on the Payment

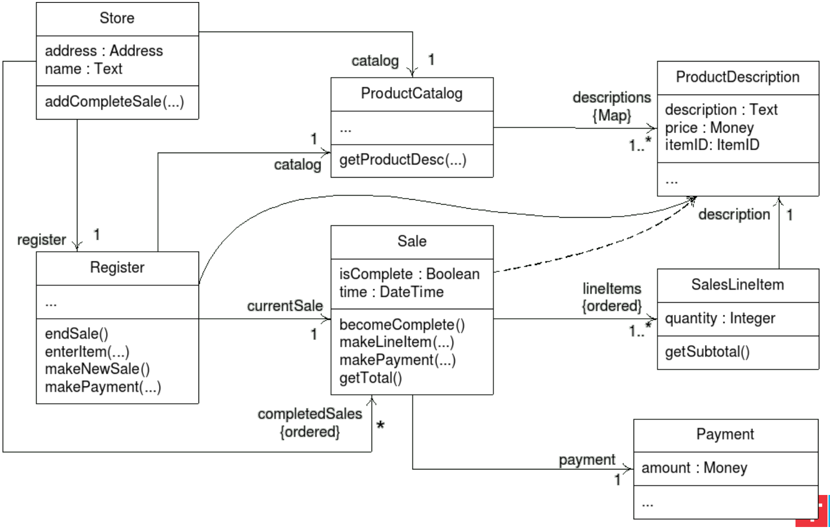
- A balance must be displayed/printed in GUI. The question is, who will be assigned with the responsibility to compute the balance.
- This responsibility can be assigned to
    - Sale – it knows a total amount for the sale,
    - Payment – it knows a received payment (cash).
- Sale class is the right choice because it knows a Payment object, not that the Payment knows the Sale.
  (otherwise, we would need to add a new association/dependency from the Payment to the Sale, i.e., to increase cohesion)

getBalance Operation in a Communication Diagram



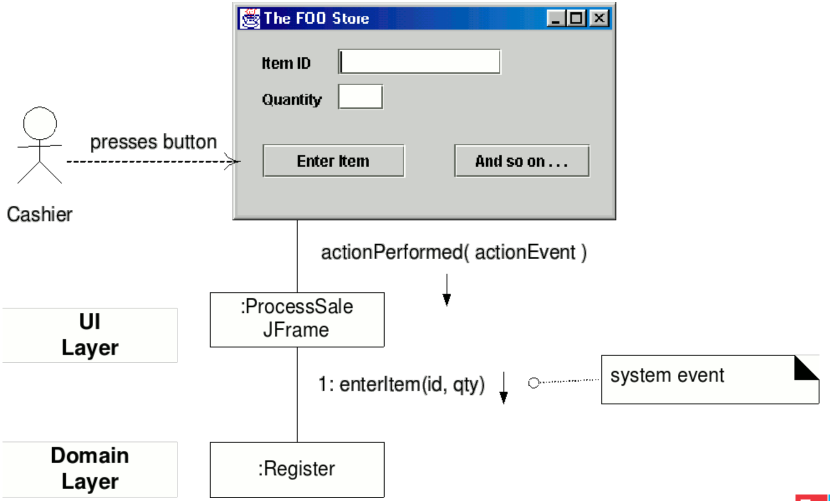
(adopted from “Applying UML and Patterns” by Craig Larman)

Classes in the Design Model



(adopted from “Applying UML and Patterns” by Craig Larman)

enterItem Operation in Graphical User Interface

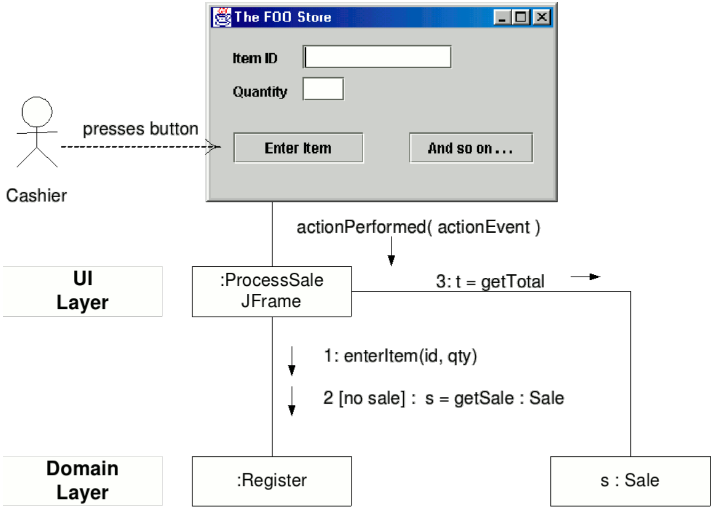


(adopted from “Applying UML and Patterns” by Craig Larman)

getTotal Operation in Graphical User Interface

- The total amount needs to be presented after adding a new item.
  (i.e., getTotal operation need to be executed after each enterItem opperation)
- This responsibility can be assigned to
  - Register – it keeps a low coupling to GUI, however, the Register interface will be bigger and its cohesion decreased,
  - a GUI class – the coupling to GUI will be increased, however, it may work if the Sale class is stable.
- Assigning the responsibility to a GUI class is the easiest way.

getTotal Operation in Graphical User Interface



(adopted from “Applying UML and Patterns” by Craig Larman)

Start Up: System Initialisation

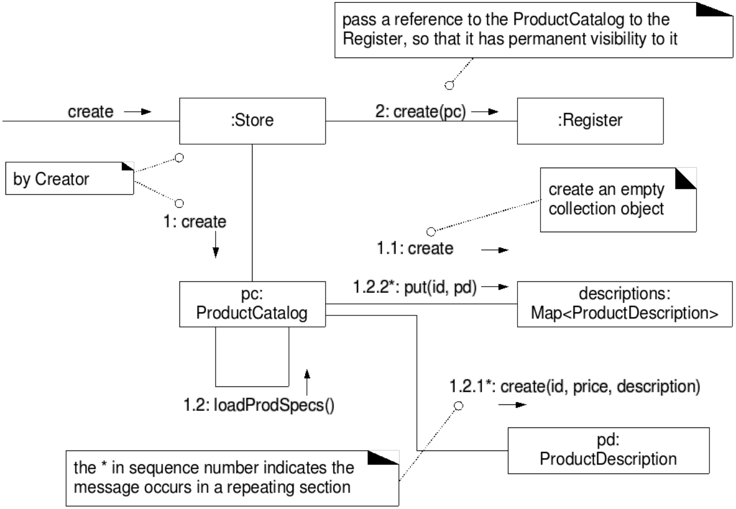
- Usually, there is an explicit or implicit start-up use case.
  (it should not be omitted in use case models)
- It should be designed later, when we will know what to initialise.
- Usually, the initialisation creates one or more domain objects.
  (directly from the static “main” method, or using factories)
- The first domain object(s) then creates another necessary objects.
- The first domain object should be a root of an aggregation/composition hierarchy.
  (then, it will create its members/components as necessary)
- **NextGen POS**: the first will be both
  - Register – it is a controller,
  - Store – it contains all the necessary objects.
    (associated with the Store object; with the high cohesion)

Start Up: Store.create Operation

To fulfil needs of the system design, the initialisation should create a Store instance and in its “create” operation, it should

- create instances of Register, ProductCatalog, and ProductDescription for all items in the catalogue,
- associate the ProductCatalog instance with instances of ProductDescription,
- create an association between Store and Register,
- create an association between Register and ProductCalatog.

create Operation in a Communication Diagram


pass a reference to the ProductCatalog to the Register, so that it has permanent visibility to it
create
:Store
2: create(pc)
:Register
by Creator
1: create
create an empty collection object
1.1: create
1.2.2*: put(id, pd)
pc: ProductCatalog
descriptions: Map<ProductDescription>
1.2: loadProdSpecs()
1.2.1*: create(id, price, description)
pd: ProductDescription
the * in sequence number indicates the message occurs in a repeating section

(adopted from “Applying UML and Patterns” by Craig Larman)

Thank you for your attention!

Marek Rychlý
<rychly@fit.vutbr.cz>