So Far …

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code

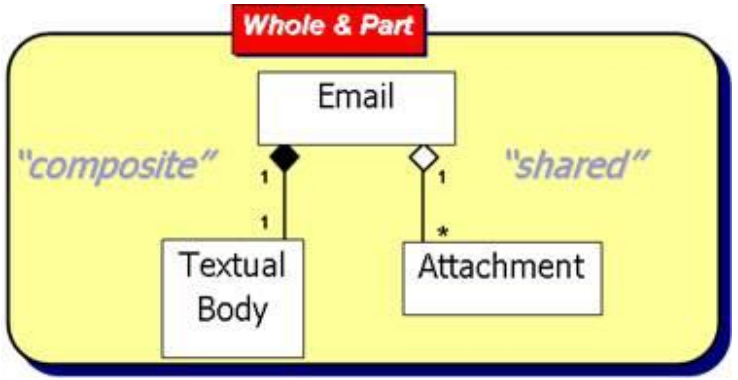
There Are Nine GRASP Patterns

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Indirection
7. Pure Fabrication
8. Polymorphism
9. Protected Variations

1. Creator

Problem	***Who should be responsible for creating a new instance of some class?***
Solution	Assign class B the responsibility to create an instance of class A if one of the following is true (the more the better): 1. B contains or aggregates A 2. B has the initializing data for A (when A is created) 3. B records A 4. B closely uses A

Aggregate Relationship



Creator

Object creation is a common task

The basic intent is to find a creator that needs to be connected to the created object

Choosing this object supports low coupling

Creator

Sometimes a creator can be identified by looking for a class that has the initializing data that will be passed in during creation

This is an example of the Information Expert pattern

If creation has significant complexity, it can be delegated to a helper class, *Abstract Factory*, *Factory Method*

Creator

Benefits:

- Low coupling is supported

Related Patterns or Principles:

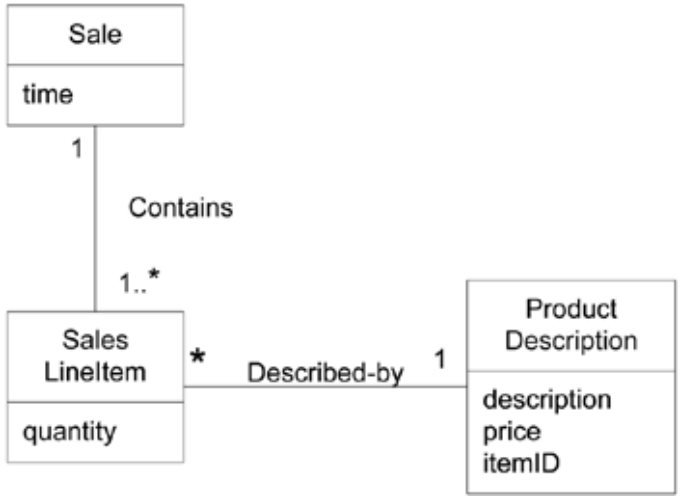
- Low coupling, a created class is likely to be **visible** to the creator class due to considerations that motivated the choice as creator
- *Factory Method*, *Abstract Factory*

Creator —POS

In the POS application, who should be responsible for creating a ***SalesLineItem*** *instance?*

Using the Creator principle, we should look for a class that aggregates and contains **SalesLineItem** instances

Partial Domain Model

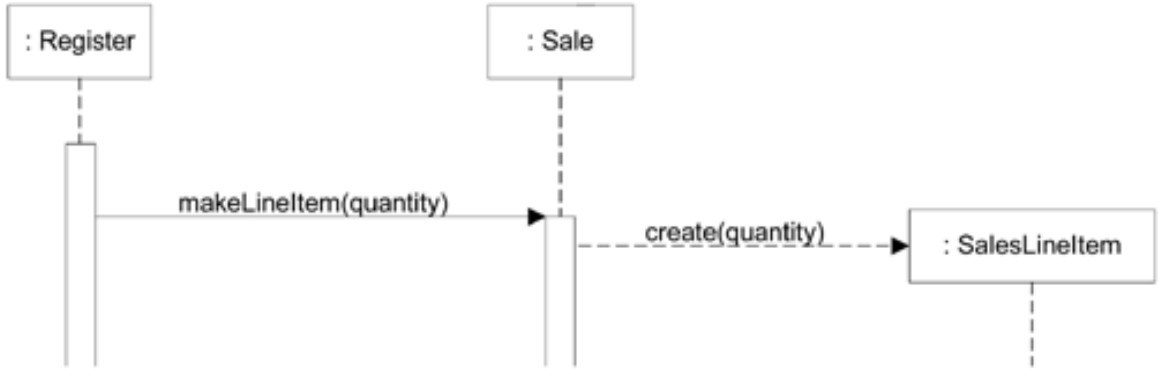


Creator –POS

Sale contains many **SalesLineItem** objects

Creator pattern recommends that **Sale** is a good candidate to have the responsibility of creating **SalesLineItem** instances

Creating a SalesLineItem



Creator —POS

This assignment of responsibilities requires that a ***makeLineItem*** method be defined in **Sale**

Note that the context in which this was considered and decided was while drawing an interaction diagram

Responsibility assignment realized by a method addition

2. Information Expert

Problem	**What is a general principle of assigning responsibilities to objects?**
Solution	Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility

Information Expert

Guiding principle, it expresses the **common intuition**

Objects should do things related to the information they have

Information expert reflects the real world,

e.g. in business who should be responsible of the profit-and-loss statement?

Answer: The person who has access to the data to create it, the chief financial officer possibly

Information Expert

Expert usually leads to designs where a software object does those operations that are normally done to the real-world (**LRG**)

Information will often be **spread** across classes and collaboration will be necessary

Information Expert

Problems in Cohesion or Coupling occur

For example, saving a **Sale** to a database, *should the **Sale** save itself?*

Probably not, see *Pure Fabrication*.

Information Expert

Benefits:

- Information Encapsulation is maintained since objects use their own information to fulfill tasks
- **Behavior is distributed across the classes** that have the information encouraging more **cohesive**, **lightweight classes** that are easier to understand and maintain

Related Patterns or Principles:

- *Low Coupling*
- *High Cohesion*

Information Expert —POS

In the POS application some class will need to know the grand total of a sale

Question: *To assigning a responsibility Who should be responsible for knowing the grand total of a sale?*

Answer: The Information Expert suggests that we should look for a class has the information needed to determine the total

Information Expert —POS

Question: *Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?*

Recall: The Domain Model illustrates conceptual classes of the real-world domain, and the Design Model illustrates software classes

Answer: If there are relevant classes in the Design Model, look there first, if not, look in the Domain Model and attempt to use what is present to inspire the creation of corresponding design classes

Information Expert —POS

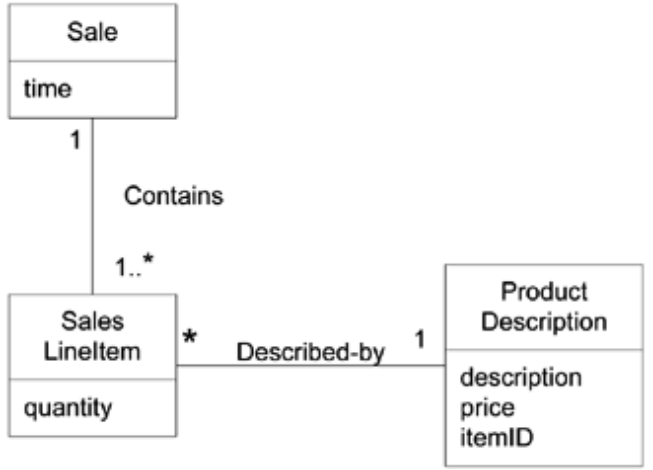
We look to the Domain Model for information experts and see the real-world **Sale** as a candidate

We add a **Sale** software class to the Design Model, if we haven't already

Give it responsibility of **knowing** its total with method named, perhaps, ***getTotal***

Also, this approach gives a low representational gap

Associations of Sale (Domain Model)



Partial Interaction, Class Diagrams Design Model

Information Expert —POS

What information is needed to determine the grand total?

It is necessary to know all the **SalesLineItem** instances of a sale and the sum of their subtotals

A **Sale** instance contains the **SaleLineItems** and so by the guideline of Information Expert

Sale is a suitable class for this responsibility

A Receipt!

Berghotel
Grosse Scheidegg
3818 Grindelwald
Familie R.Müller

Rech.Nr. 4572 30.07.2007/13:29:17
Bar Tisch 7/01

2xLatte Macchiato	à	4.50	CHF	9.00
1xGloki	à	5.00	CHF	5.00
1xSchweinschnitzel	à	22.00	CHF	22.00
1xChässpätzli	à	18.50	CHF	18.50

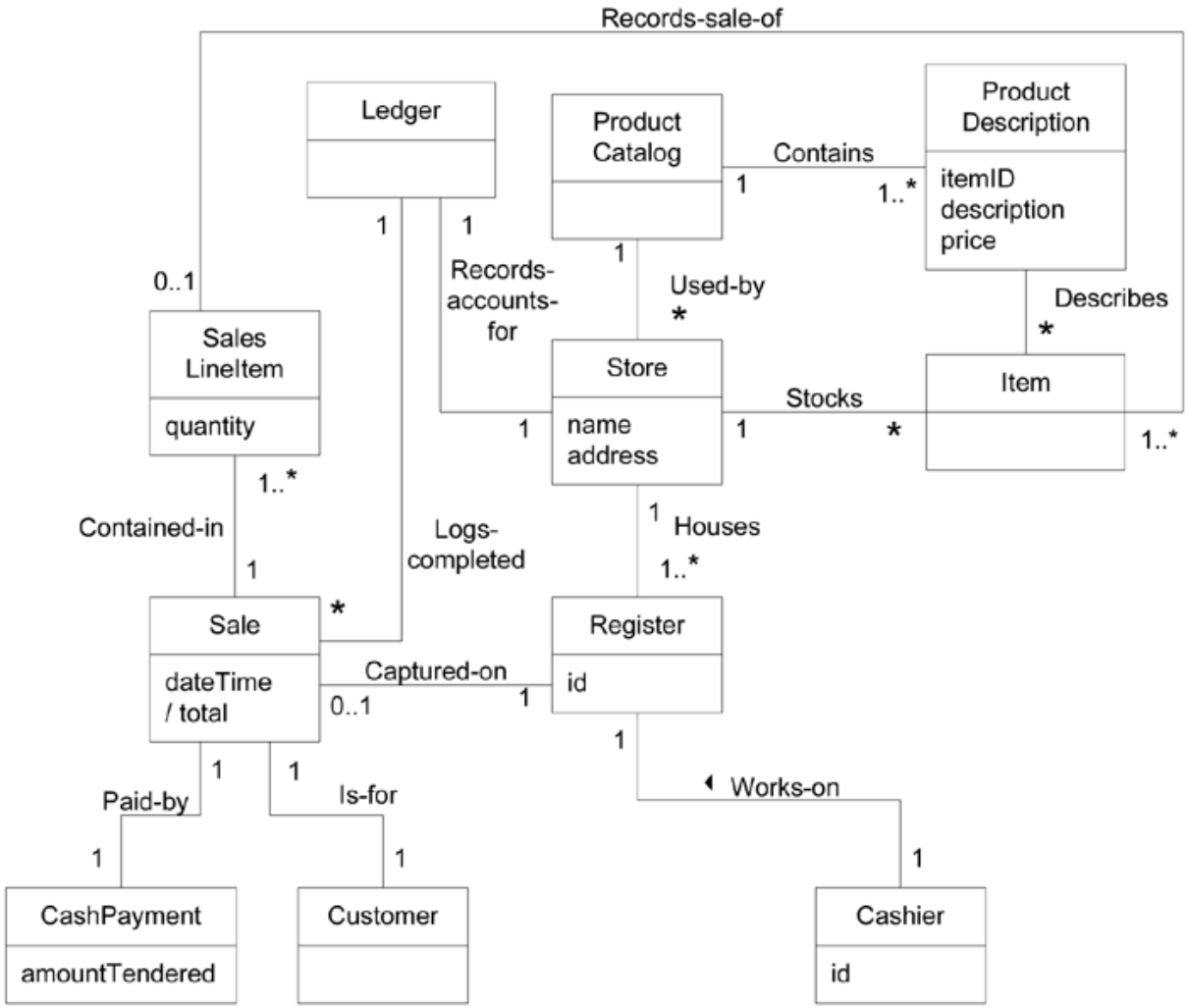
Total : CHF **54.50**

Incl. 7.6% MwSt 54.50 CHF: 3.85

Entspricht in Euro 36.33 EUR
Es bediente Sie: Ursula

MwSt Nr.: 430 234
Tel.: 033 853 67 16
Fax.: 033 853 67 19
E-mail: grossescheidegg@bluewin.ch

POS Partial Domain Model



Information Expert —POS

*What information is necessary to determine the **line-item subtotal**?*

SalesLineItem.quantity**,**

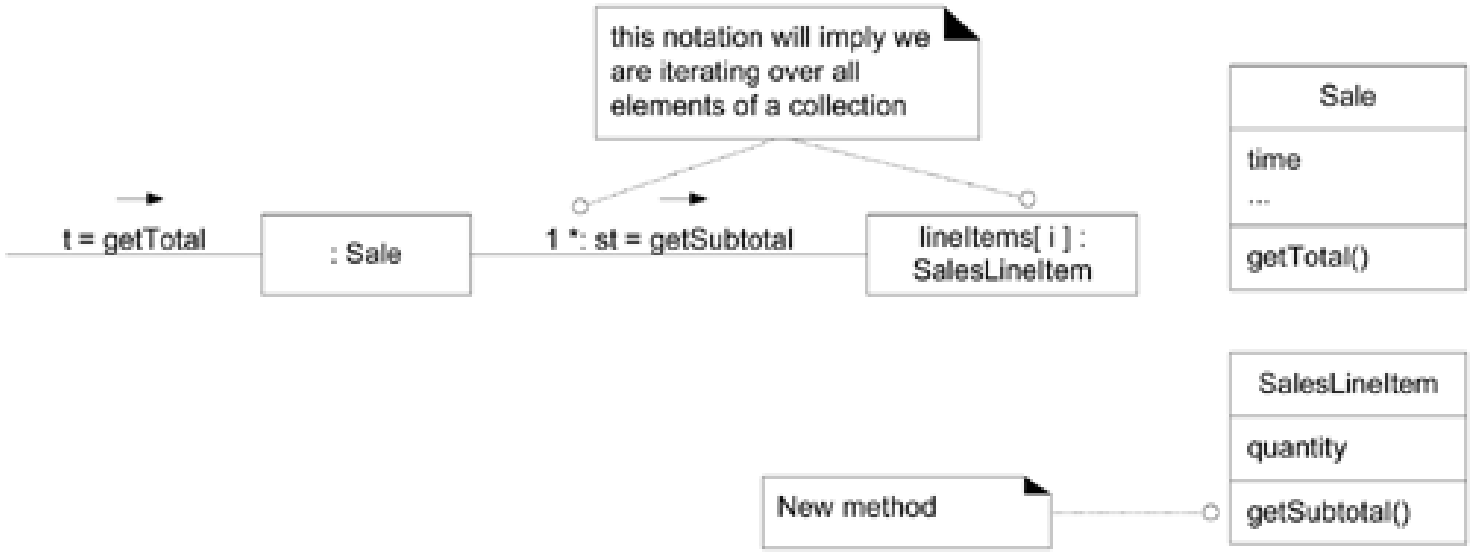
ProductDescription.price are necessary

SalesLineItem knows its quantity and its associated **ProductDescription**,

and so **SalesLineItem** should determine the subtotal

SalesLineItem it is the information expert, ***getsubtotal***

Information Expert —POS



Calculating the *Sale* total

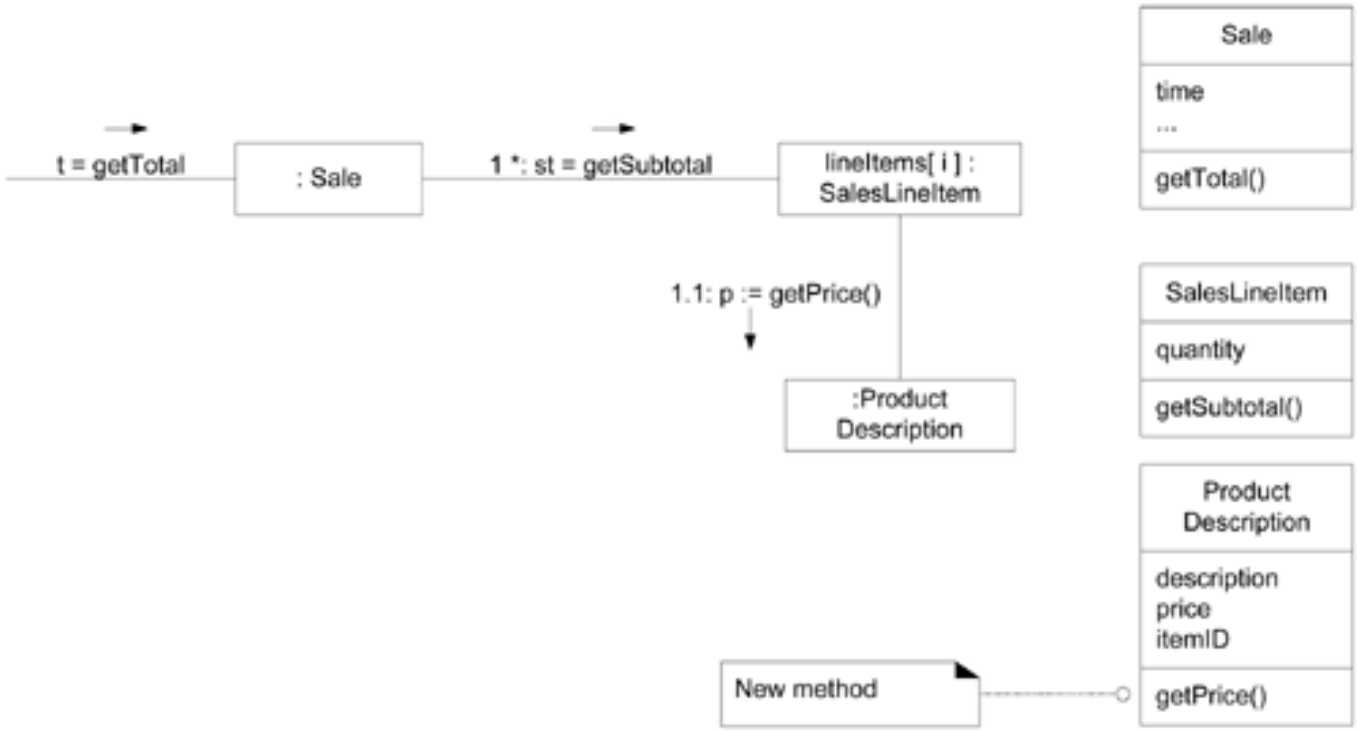
Information Expert —POS

What about the price of the product?

ProductDescription is an information expert regarding product price, it knows!

therefore, **SalesLineItem** sends **ProductDescription** a message asking for the product price

Information Expert —POS



Calculating the *Sale* total

Information Expert —POS

To fulfill the responsibility of knowing and answering the sale's total, we assigned three responsibilities to three design classes of objects as follows

Design Class	Responsibility (knowing)
Sale	knows sale total
SalesLineItem	knows line-item subtotal
ProductDescription	knows product price

Information Expert - Comments

Information Expert is a basic guiding principle that frequently used in the assignment of responsibilities and object design

Information Expert expresses the common "intuition" that objects do things related to the information they have,

But too much "intuition“ make cause problems, will see!
SOLID?

Information Expert —Animation principle

The fulfillment of a responsibility often requires information that is spread across different classes of objects

Many "partial" information experts collaborate in the task

- Sales total problem ultimately required the collaboration of three classes of objects
- Whenever information is spread across different objects, they will need to interact via messages to share the work

Information Expert —Animation principle

Software object does those operations to the inanimate real-world thing it represents

In OO, all software objects are "**alive**" or "**animated**," and they can take on responsibilities and do things

They do things related to the information they know

The "animation" principle in object design; it is like being in a cartoon where everything is alive.

3. Low Coupling

Problem	***How to support low dependency, low change impact, and increased reuse?***
Solution	Assign a responsibility so that (unnecessary) coupling remains low Evaluative, use this principle to evaluate alternatives

What is Coupling Anyway?

TypeX has an attribute that refers to a **TypeY** instance or **TypeY** itself

A **TypeX** calls on services of a **TypeY** object

TypeX has a method that references a **TypeY** instance or **TypeY** itself by any means; parameter, local variable, return object

TypeX is a direct or indirect subclass of **TypeY**

TypeY is an interface, and **TypeX** implements that interface

Low Coupling

Benefits:

- Un-coupled components are not affected by changes in each other
- Simpler to understand in isolation
- Easier reuse and test

Related Patterns or Principles:

- *Protected Variation*

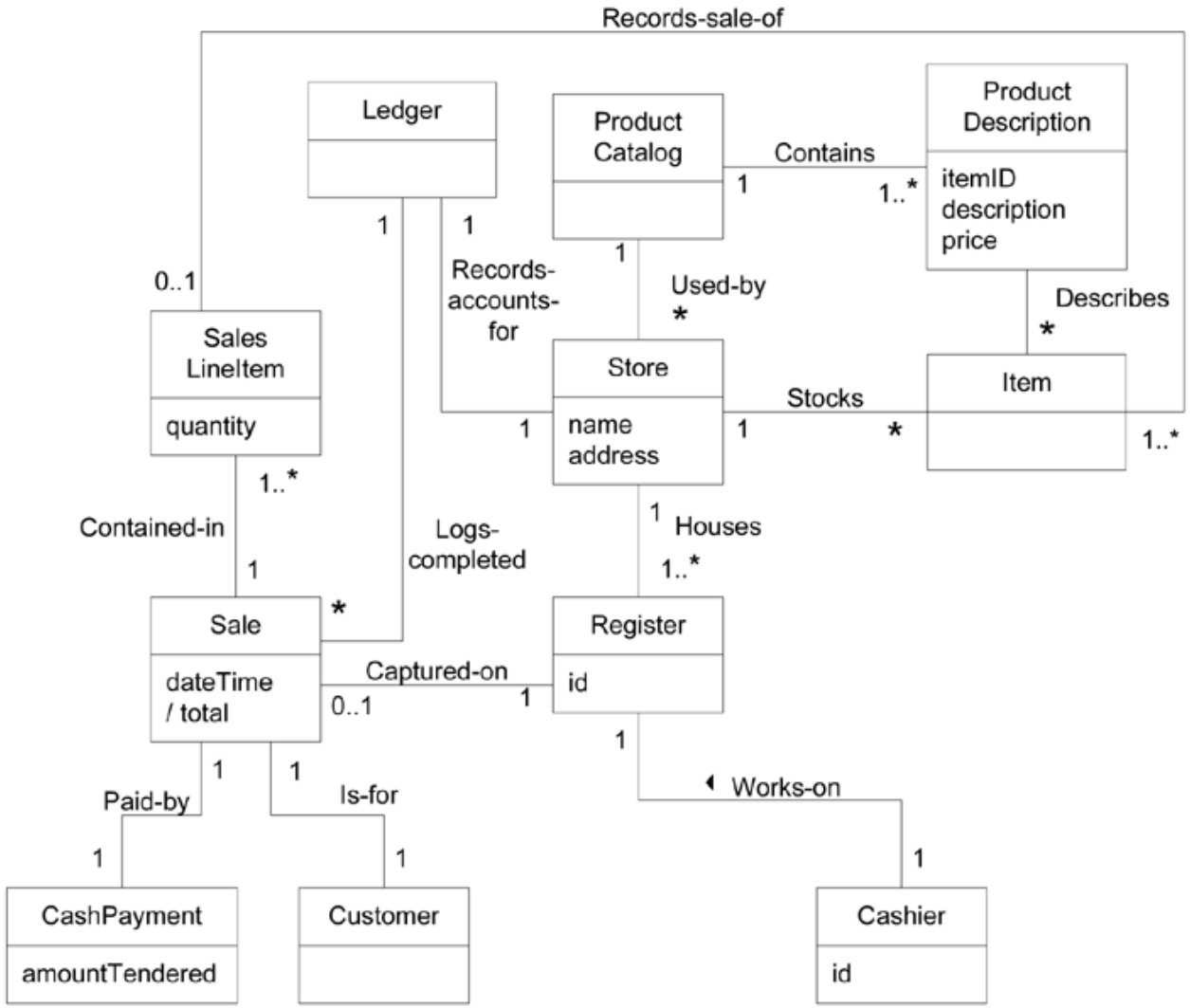
Low Coupling —POS

*What class should create a **Payment** instance and associate it with the **Sale**?*

Creator suggests **Register** since a **Register** "records" a **Payment** in the real-world domain (**LRG**)

This assignment of responsibilities couples the **Register** class to knowledge of the **Payment** class

POS Partial Domain Model

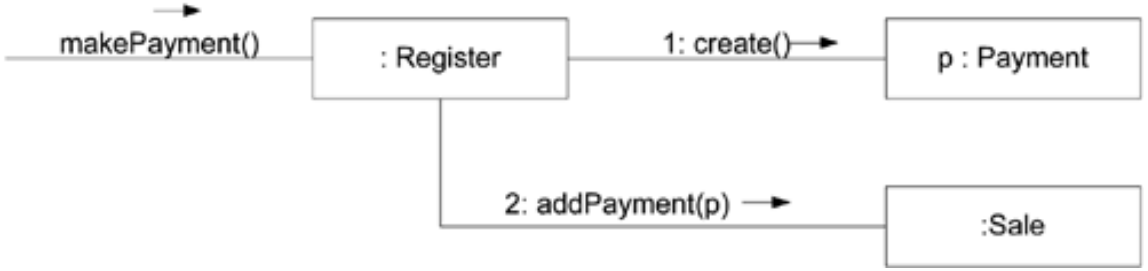


Low Coupling —POS

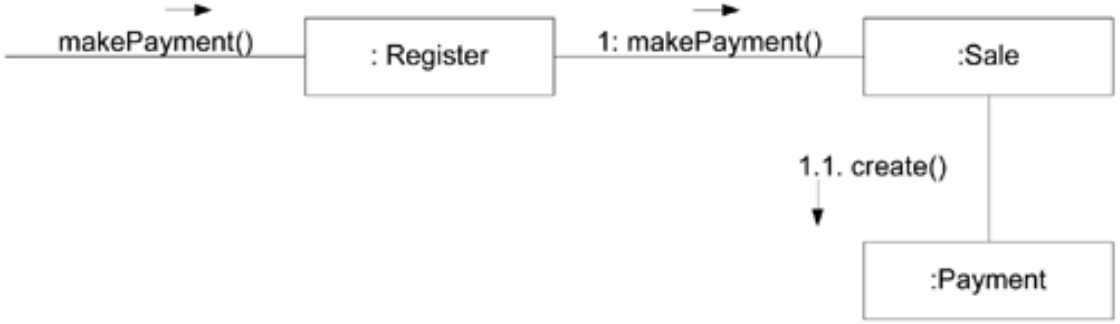
Low coupling suggests **Sale** should create **Payment**.

Which is best?

The second design gives lower coupling and, other things being equal, is probably preferred



Register creates Payment



Sale creates Payment

Low Coupling —Comments

Low Coupling is a principle to keep in mind during all design decisions

Low Coupling is an underlying goal to continually consider while evaluating design decisions

In general, classes that are inherently generic in nature and with a high probability for reuse should have especially low coupling

Low Coupling —Comments

Low Coupling taken to **excess** yields a poor design

A design with a few in-cohesive, **bloated**, and complex active objects that do all the work and with many passive low-coupled objects that act as simple data repositories

Low Coupling is an **evaluative** guideline

4. Controller

Problem	***What first object beyond the UI layer receives and coordinates ("controls") a system operation?***
Solution	Assign the responsibility to a class representing one of the following choices: 1. Represents the overall "system", a "root object", a "device" the software is running within, or a major "subsystem" (these are all variations of the **Facade Controller**) 2. Represents a use case scenario within which the system event occurs, (a use-case session controller)

Controller

This is a delegation pattern

UI, user interface, layer (boundary objects) shouldn't contain application logic

UI objects must delegate work requests to another layer

The controller pattern summarizes the common choices for the object to receive the work request

Controller

A common defect of controllers results from over-assignment of responsibility “**Bloated**" controllers

Solution: more controllers are needed or the controller is not delegating enough

Controller

Benefits:

- Increased potential for reuse and pluggable interfaces
- Opportunity to reason about the state of the use case

Related Patterns or Principles:

- *Command Pattern*
- *Facade Pattern*
- *Layers*
- *Pure Fabrication*

Controller —POS

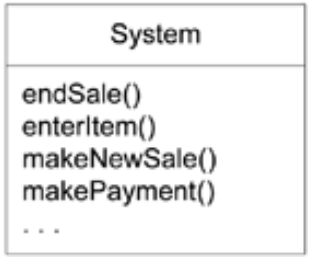
The POS application contains several system operations

During analysis (e.g. **SSD**), system operations may be assigned to the class System, to indicate they are **System** operations

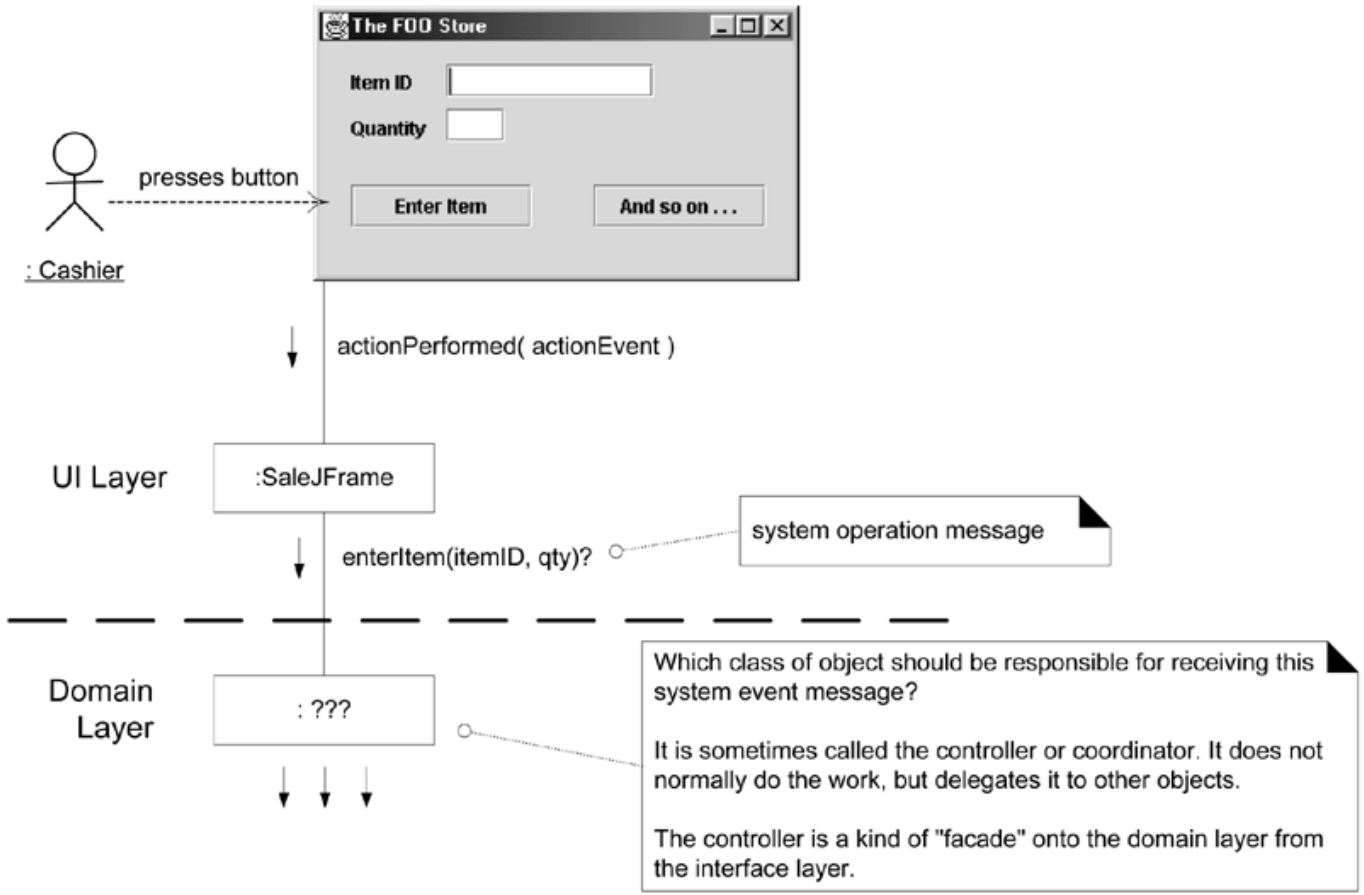
There will not be software class named System though

During design, a controller class is assigned the responsibility for system operations

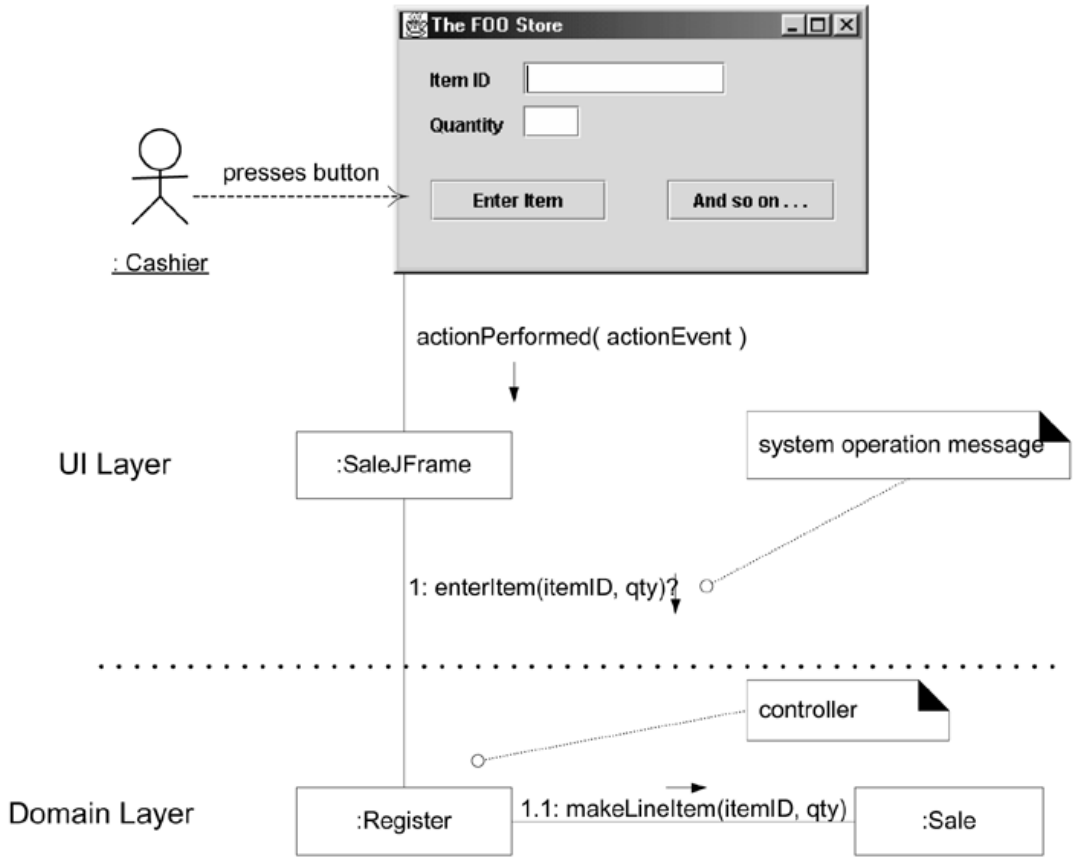
Some system operations of the NextGen POS application



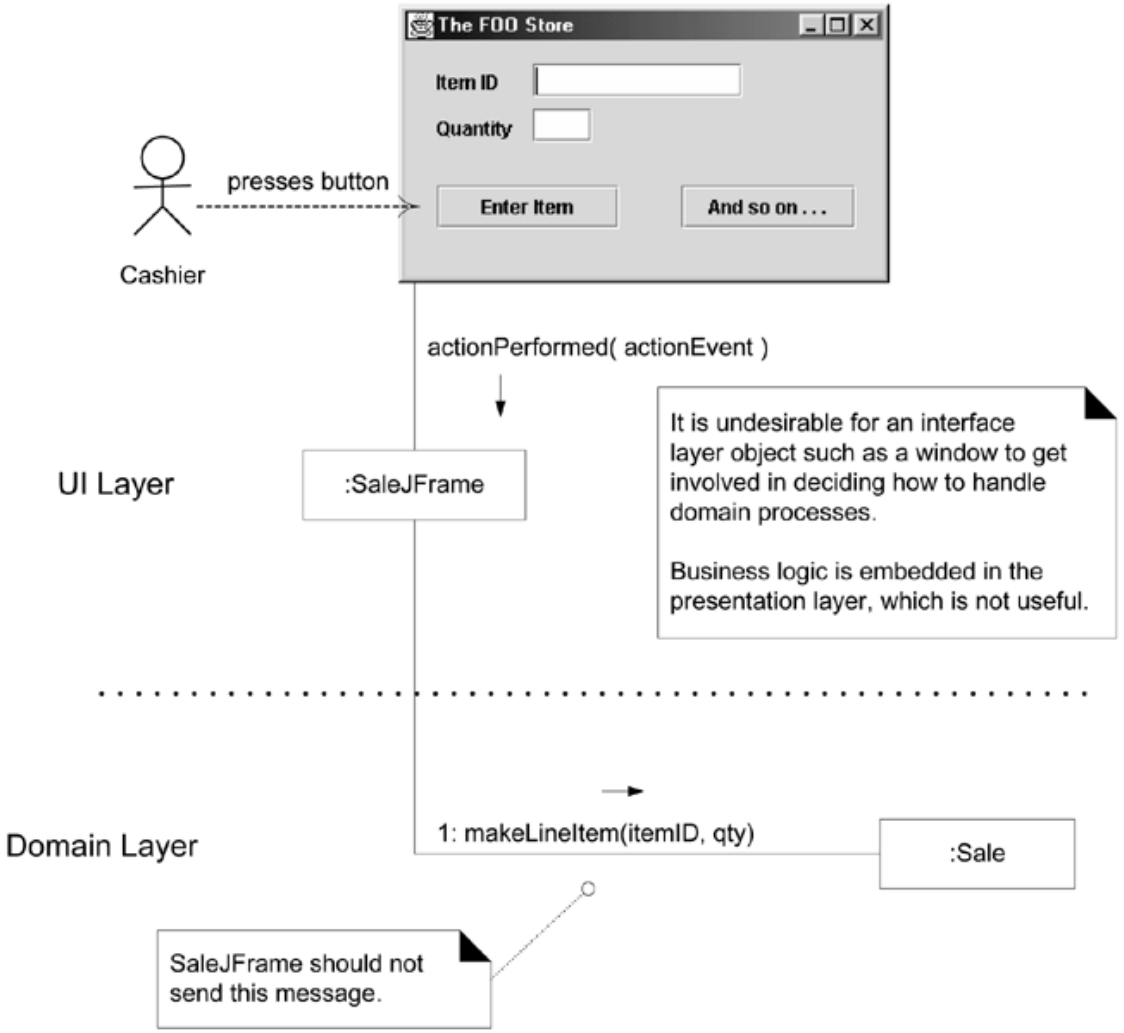
What object should be the Controller for enterItem?



Desirable coupling of UI layer to Domain layer: Device Rep.

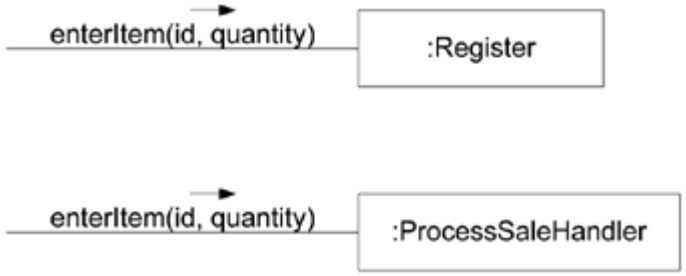


Less desirable coupling of interface layer to domain layer

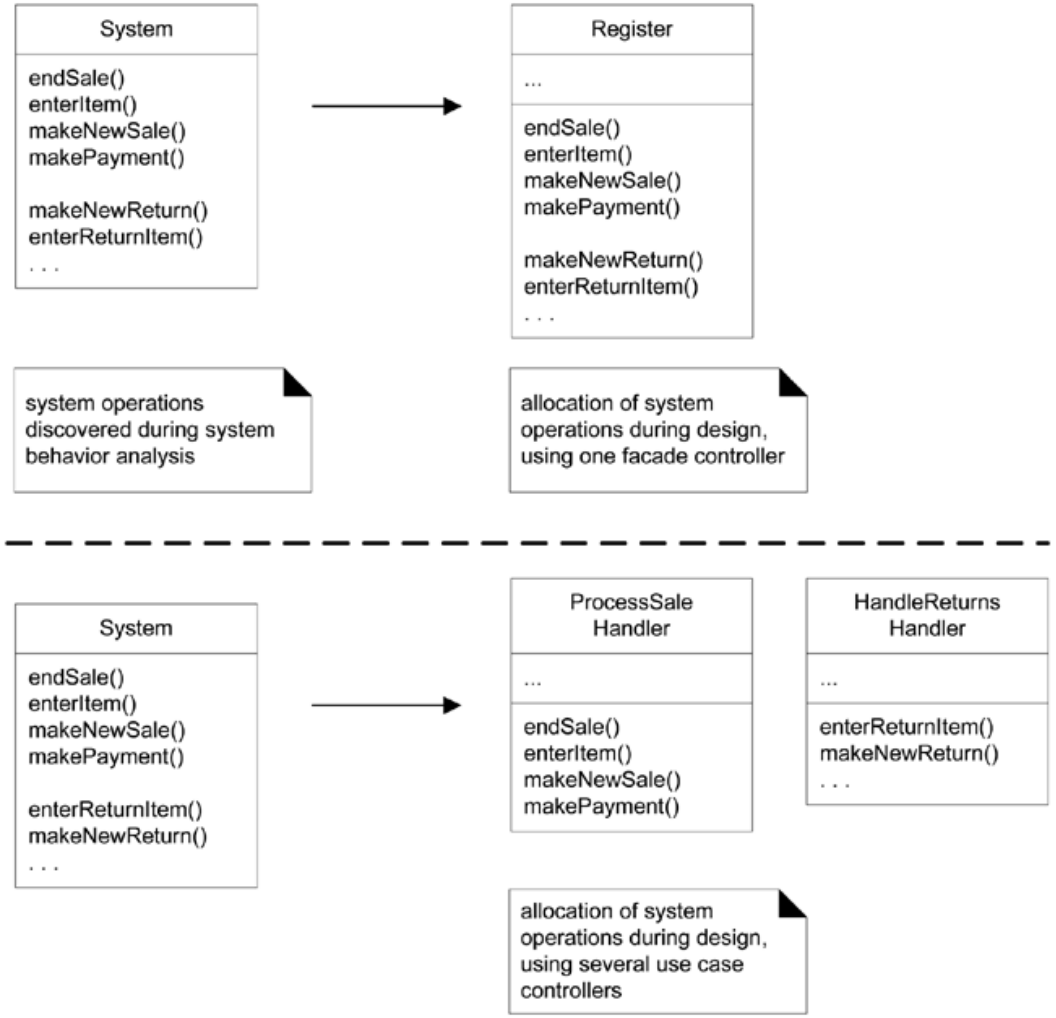


What object should be the Controller?

Controller choices



Allocation of system operations



5. High Cohesion

Problem	**How to keep objects focused, understandable, and manageable, and, as a side effect, support low coupling?**
Solution	Assign responsibilities so that cohesion remains high Evaluative, use this principle to evaluate alternatives

High Cohesion

Coupling and Cohesion impact each other

Consider a class **A** that does two logically different things

A is coupled to the resources necessary to accomplish the two different things

If **A** were split into two classes each performing one of the logically different tasks

Each would only be coupled to the classes necessary to accomplish its task

High Cohesion —Rule of Thumb

- A class with high cohesion has a relatively small number of methods,
- with highly related functionality,
- and does not do too much work. **S**OLID?
- It collaborates with other objects to share the effort if the task is large

High Cohesion —Analogy

A person takes on too many unrelated responsibilities—especially ones that should properly be delegated to others—then the person is **not effective**

This is observed in some managers who have not learned how to delegate.

These people suffer from low cohesion; they are ready to become "unglued." needs to **refocus!**

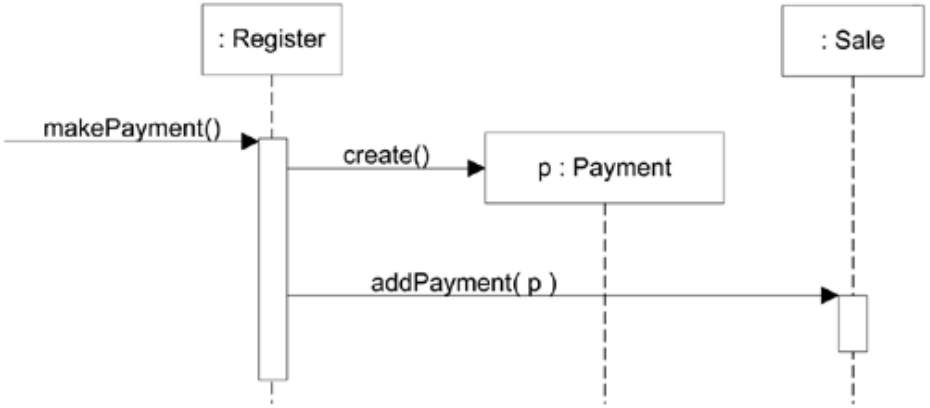
High Cohesion

Benefits:

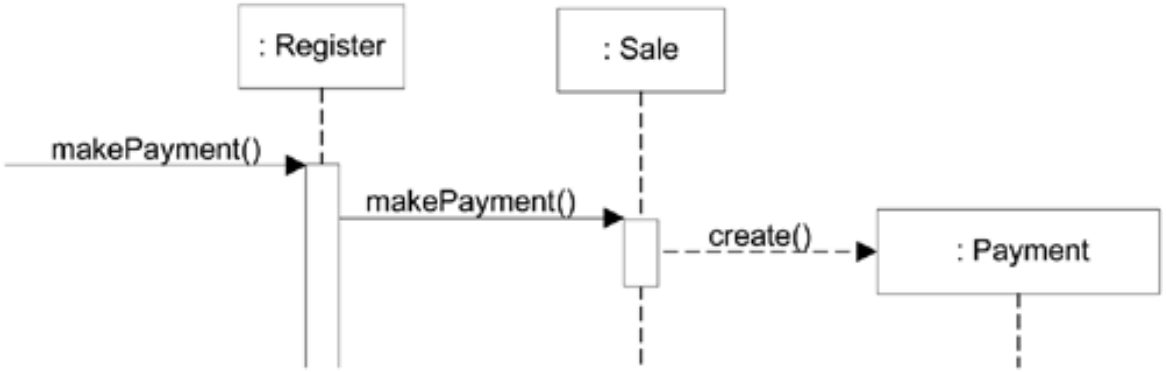
- Clarity and ease of comprehension of the design is increased
- Maintenance and enhancements are simplified
- Low coupling is often supported
- Reuse is increased

High Cohesion —POS

Register Creates Payment



Sale Creates Payment



Modular Design

Modularity is

the property of a system that has been **decomposed** into a set of **Cohesive and Loosely Coupled Modules** [Booch94].

At the basic object level, we achieve modularity by designing each **method** with a clear, single purpose and by grouping a related set of concerns into a class

So Far …

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code

Object Design Examples with GRASP

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

OOD Objective

Design Use Case realizations

Apply the GRASP patterns

Use the UML class and interaction diagram notation to illustrate the design of objects

Assignment of Responsibility

- Assignment of **responsibilities,**
- **Role**
- And **collaborations**

Either while **diagramming** or while **programming**

Use-Case Realizations

A particular use case is realized within the design model

A designer can describe one or more scenarios of a use case

Each of these is called a use case realization

Use-Case Realizations, How?

1. UML Interaction diagrams involve message interaction between software objects to illustrate use-case realizations

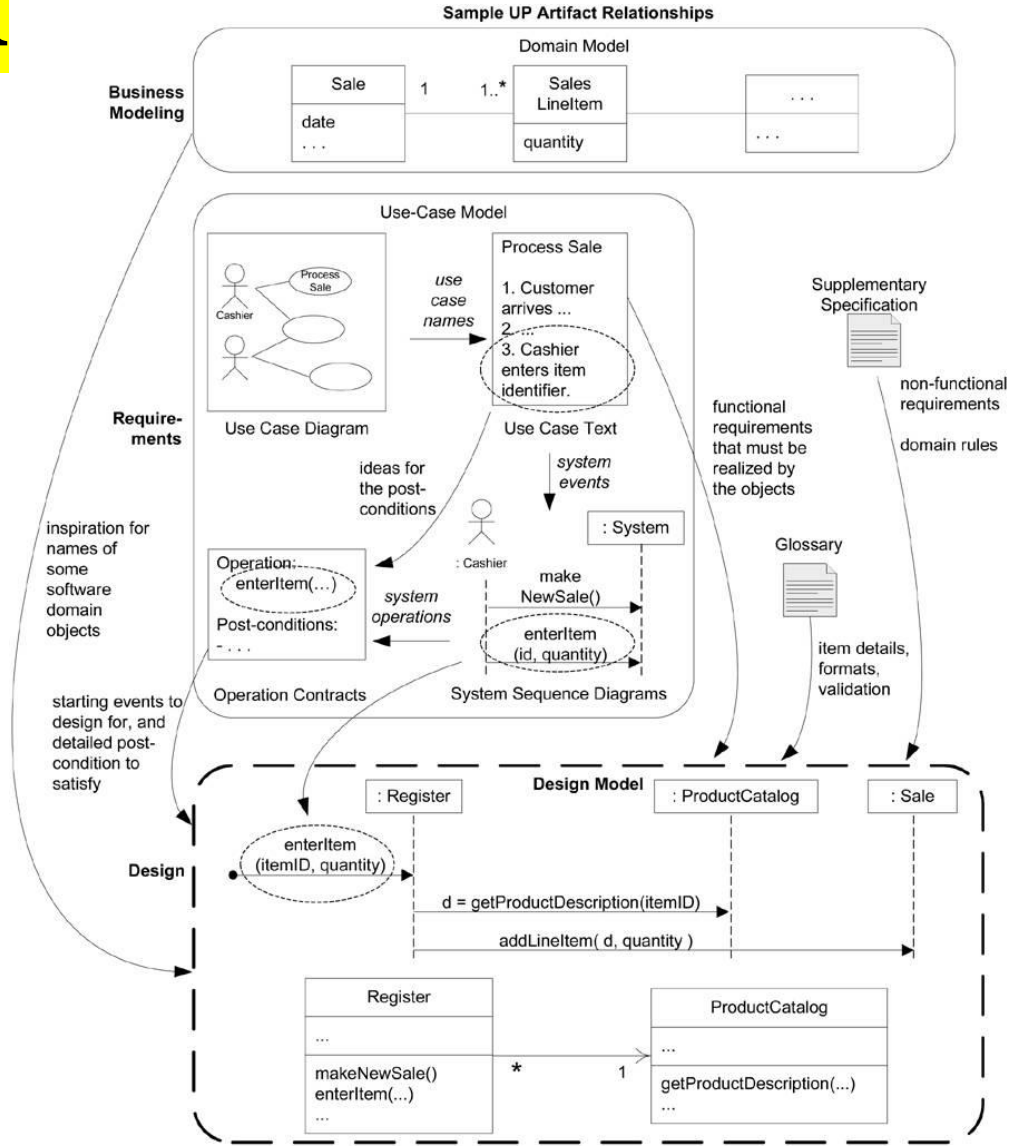
2. Principles and Patterns can be applied during this design work

3. Inspired by the names of conceptual classes in the Domain Model (**LRG**)

Use-Case Realizations

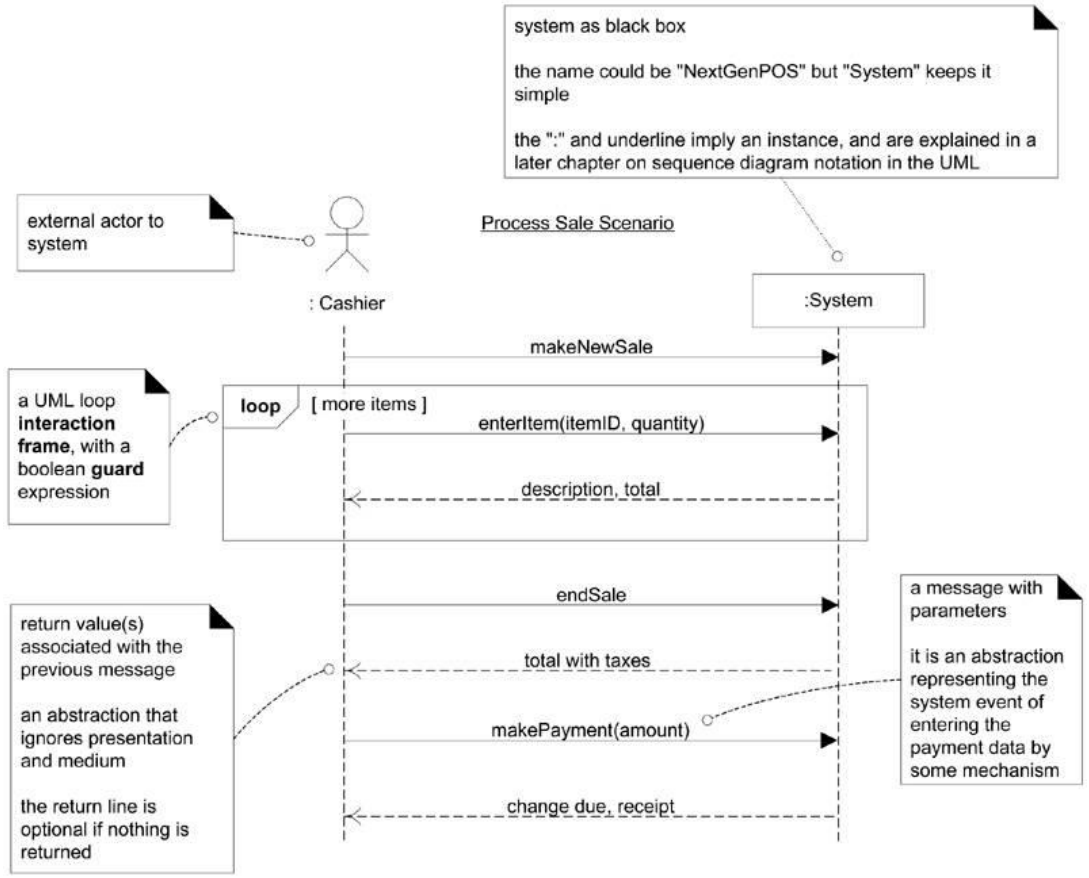
POS

Artifact Relationships, Use Case Realization



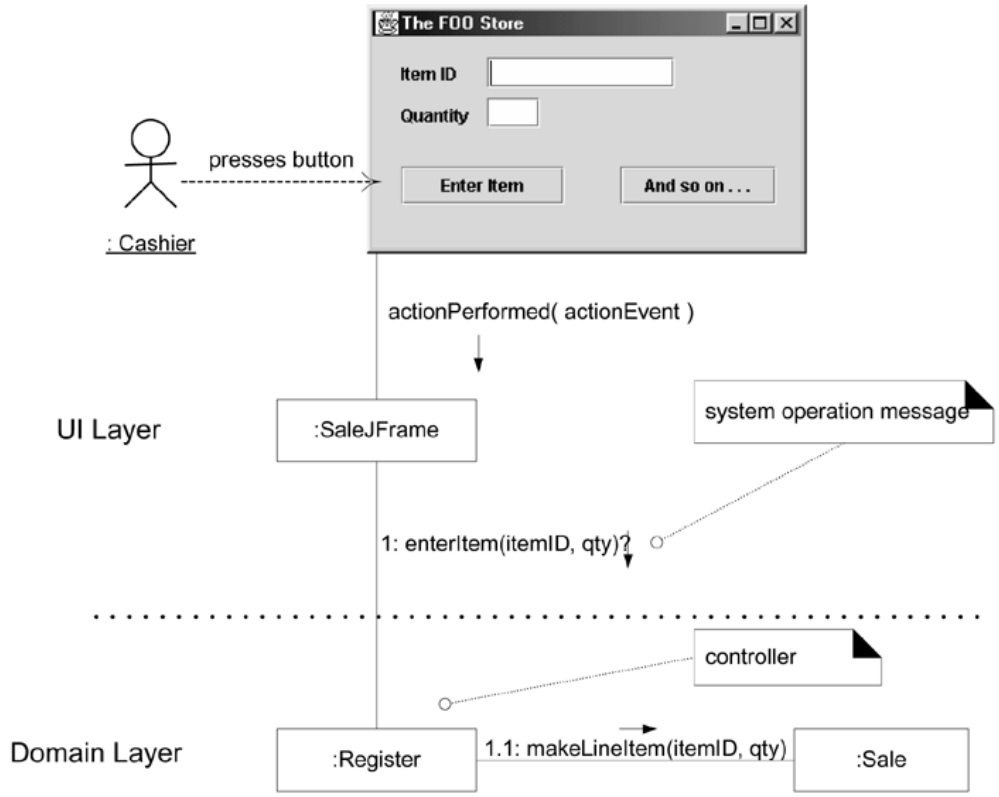
Artifact-Influence

Use case suggests the system operations that are shown in **SSDs**



Artifact-Influence

System operations become the starting messages entering the **Controllers** for domain layer interaction diagrams



Interaction Diagrams

Your System Sequence Diagram (SSD) list the system events

Create a separate Interaction or sequence diagram for each system operation under development

For each system event, make a diagram with it as the starting message

If the diagram gets complex, split it into smaller diagrams

Operation Contract Postconditions —Almost Design

Using the contract operation postconditions and use case description as a **starting point**

Interaction Diagrams and System Events

At Iteration-I of POS application, realize two use-cases and their associated System Events

Process Sale

- *makeNewSale*
- *enterItem*
- *endSale*
- *makePayment*

Start Up

- *startUp*

Initialization and the 'Start Up' Use Case

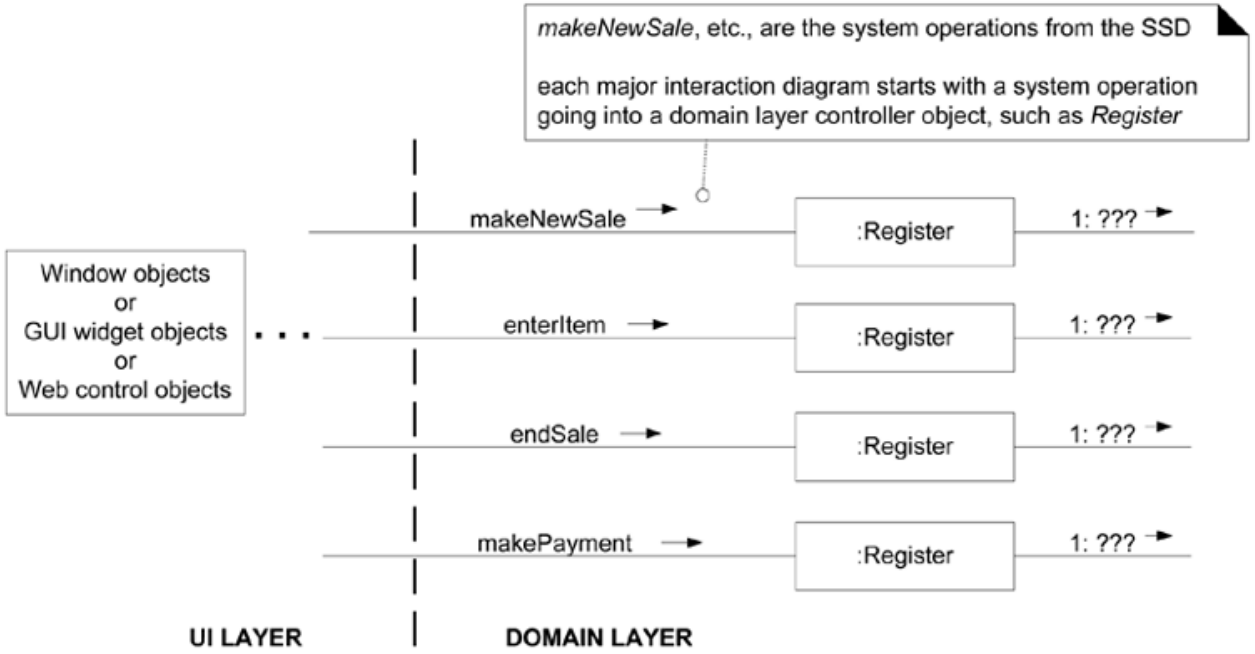
"**Start Up**" use case realization is the design context in which to consider creating most of the 'root' or long-lived objects

In coding, program at least some **Start Up** initialization first

In OOD modeling, Start Up initialization design is last

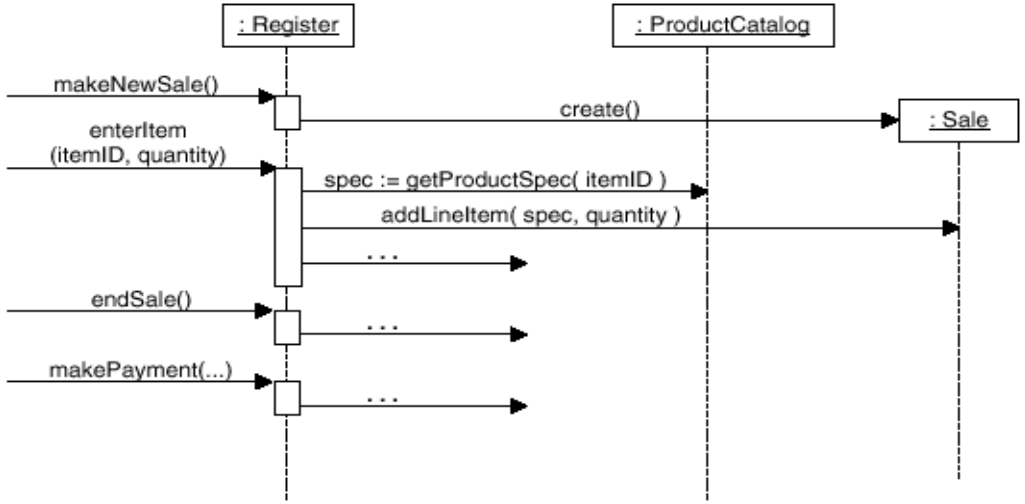
e.g., Process Sale use case realization before the supporting Start Up design

SDD to Interaction Diagram



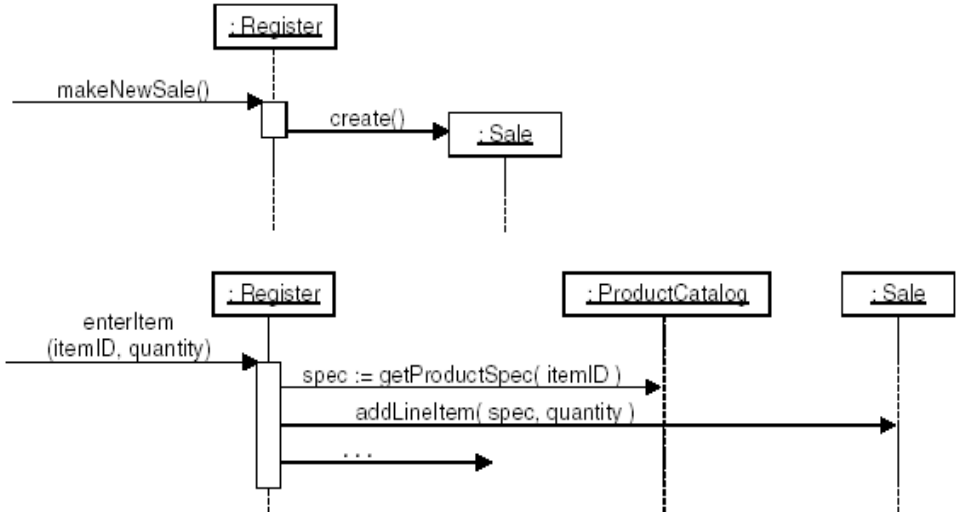
SDD to Sequence Diagrams

Sequence diagrams may be possible to fit all system event messages on the same diagram



Multiple Sequence Diagrams

Multiple sequence diagrams and system event message handling



Operation Contracts and Use-Case Realizations

It may be possible to design use-case realizations **directly** from the use-case text

SDD isolates system operations

Contracts may have been written that add greater detail or specificity

Bypass Contracts?

Encourage investigative work during

The analysis phase rather than the design phase

Requirements Are Not Perfect

Postconditions are an initial best guess or estimate of what must be achieved

Use-case and postconditions may not be accurate

Spirit of **iterative** development is to capture a

- **Reasonable degree** of **information** during **requirements analysis**
- Filling in **details** during design and implementation

Use-Case Realization —Domain Model

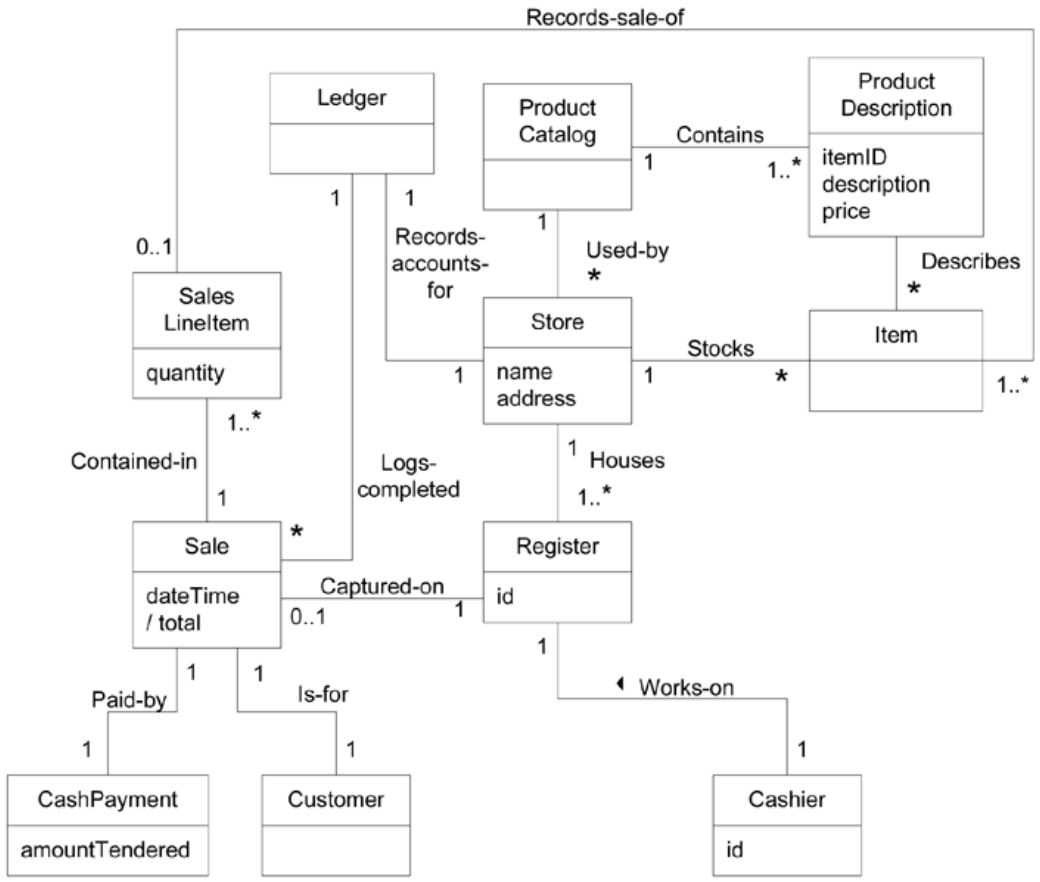
Interaction diagrams, Domain Model inspires some of the software objects, e.g., a **Sale** conceptual class and **Sale** software class (**LRG**)

Domain Model—or all analysis artifacts—**won't be perfect**; errors and omissions expected

You will discover new concepts that were previously missed, ignore concepts that were previously identified,

and do likewise with associations and attributes

POS Partial Domain Model

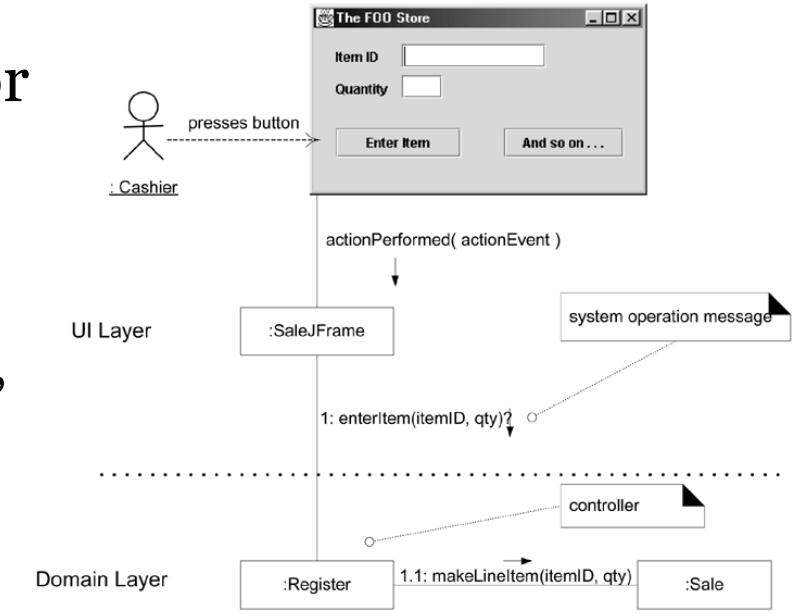


How to Design *makeNewSale*?

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 a **Register** (association formed) - Attributes of *s* were initialized

Choosing the Controller Class

Our design choice involves choosing the controller for the system operation message e.g. ***enterItem***



By the Controller pattern, some choices are a system, root, device, or subsystem:

Register, **POSSystem**

A receiver or handler of system events of a use-case scenario

ProcessSaleHandler, **ProcessSaleSession**.

Applying the GRASP Controller Pattern

A device-object facade controller **Register** is satisfactory

A few system operations and if the facade controller is not taking on too many responsibilities (not incohesive)

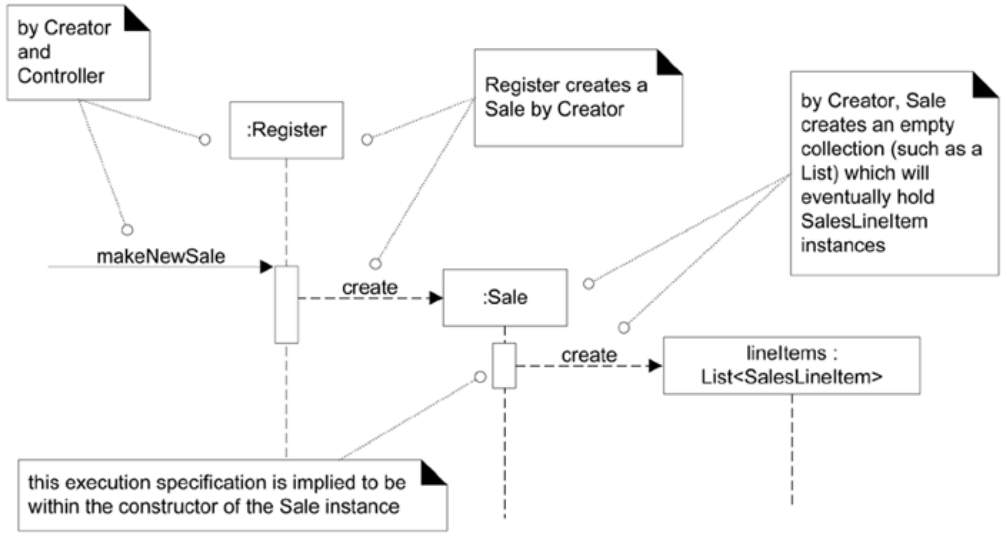
1. Creating a New Sale, 2. Asso. Register, 3. init. Sale

An empty collection is required to record all the future **SalesLineItem** instances

This collection will be contained within and maintained by the **Sale** instance, Creator

Therefore, the **Register** (device) creates the **Sale**, and the **Sale** creates an empty collection

Creating a New Sale



Creating a New *SalesLineItem*

Contract CO2: enterItem	
Operation	enterItem(itemID: ItemID, quantity: integer)
Cross References	Use Cases: Process Sale
Preconditions	There is a sale underway.
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)

1. SalesLineItem created
2. And Associated With Sale

By Creator, **Sale** is an appropriate to create a **SalesLineItem**

Associate **Sale** with newly created **SalesLineItem** by storing the new instance in its collection of line items

3. SalesLineItem.quantity =quantity

SalesLineItem needs a quantity when created

Register must pass quantity along to the **Sale**

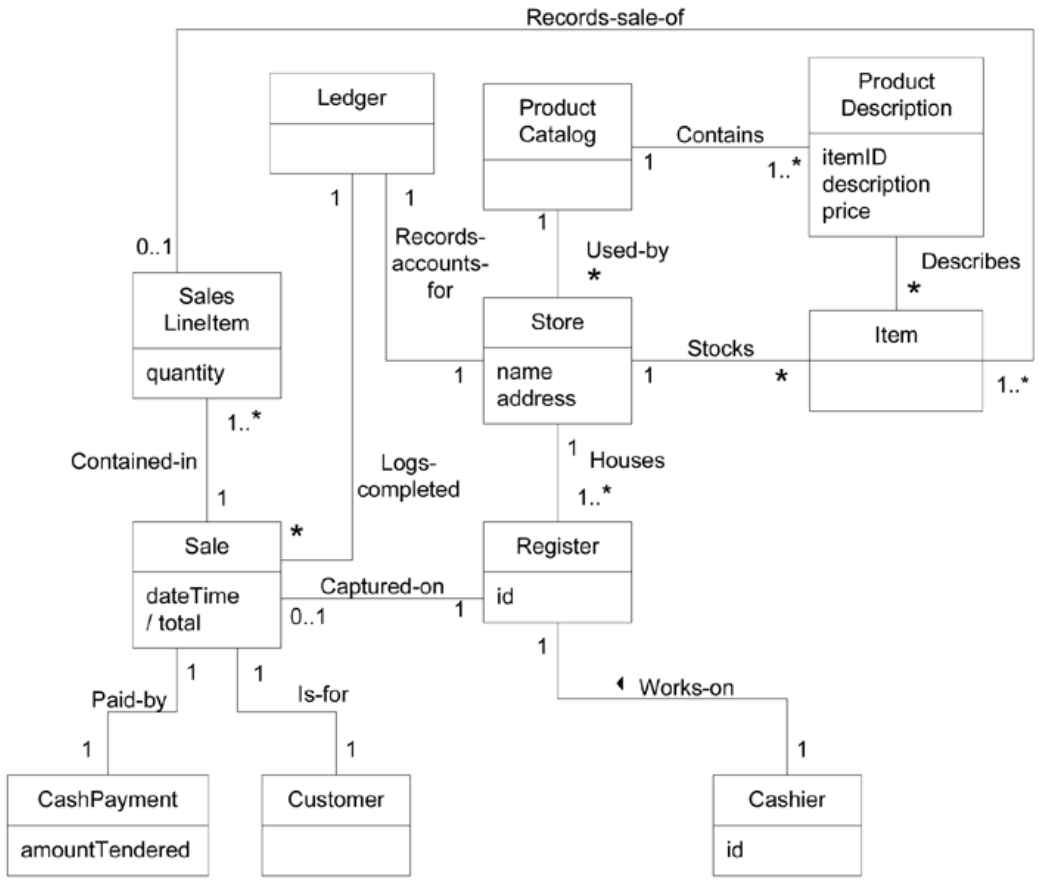
Sale must pass it along as a parameter in the create message; a constructor call with a parameter.

4. Finding a ProductDescription

SalesLineItem needs to be associated with the **ProductDescription** that matches the incoming itemID

Who should be responsible for knowing a ***ProductDescription****, based on an itemID match?*

POS Partial Domain Model



Finding a ProductDescription

Neither Creation nor choosing a Controller for a system event. *Information Expert*?

*Who knows about all the **ProductDescription** objects?*

DM reveals **ProductCatalog** contains all the **ProductDescriptions**

LRG, a software **ProductCatalog** will contain software **ProductDescriptions**, ***getProductDescription***

Expert **ProductCatalog** is a good candidate for this lookup responsibility

Visibility?

Visibility is the ability of one object to “see” or have a reference to another object

For an object to send a message to another object, it must have visibility to it

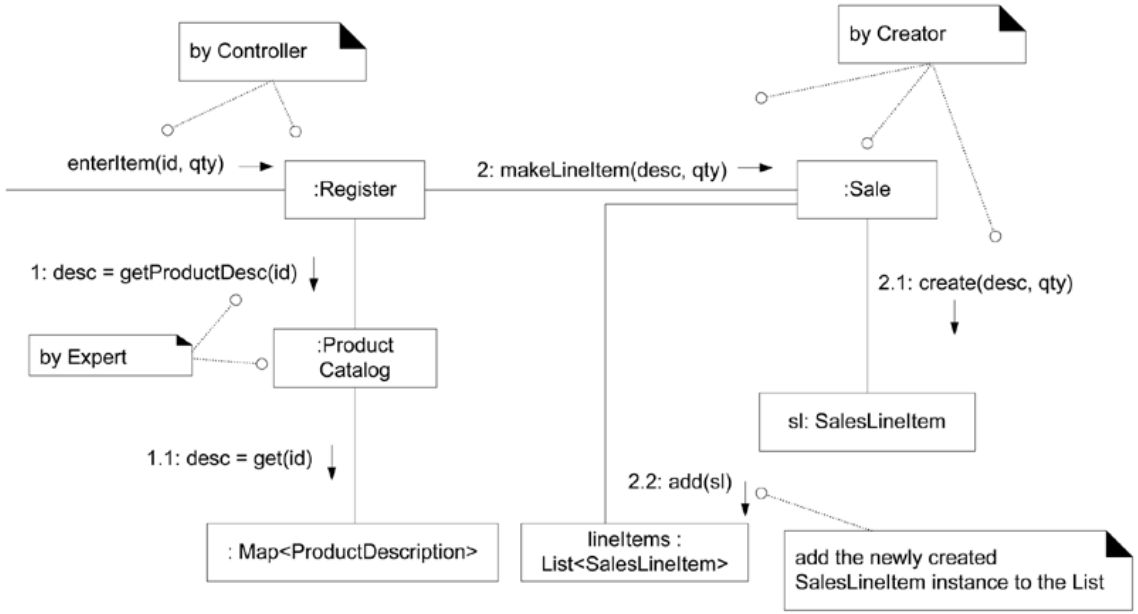
Visibility to a ProductCatalog

*Who should send the **getProductDescription** message to the **ProductCatalog** to ask for a **ProductDescription**?*

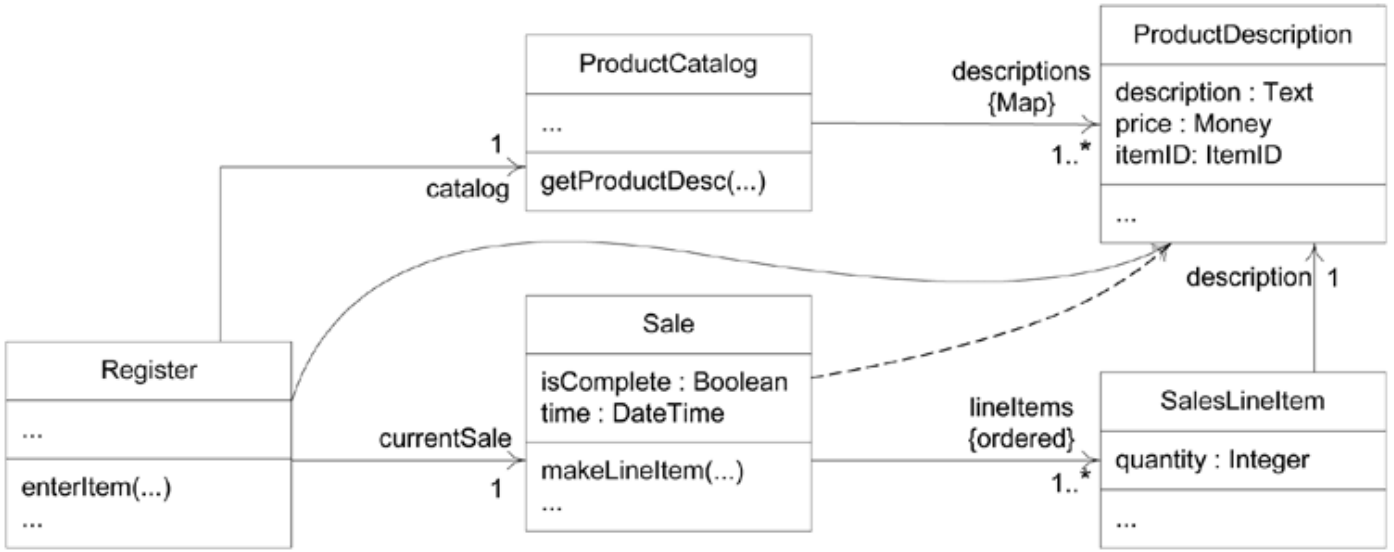
DM: **Register** object can be permanently connected to **ProductCatalog** object

Register can send the ***getProductDescription*** message to the **ProductCatalog**. That is called Visibility, Next Chapter.

enterItem, Final Design —Dynamic View



enterItem, Final Design —Static View



How to Design *endSale*?

Contract CO3: endSale	
Operation	endSale()
Cross References	Use Cases: Process Sale
Preconditions	There is a sale underway.
Postconditions	- **Sale**.isComplete became true (attribute modification)

Choosing the Controller Class

Our first choice involves handling the responsibility for the system operation message ***endSale***

Based on the Controller GRASP pattern, as for ***enterItem***, we will continue to use **Register** as a controller

Setting the Sale.isComplete Attribute

Sale.isComplete became true (attribute modification)

*Who should be responsible for setting the isComplete attribute of the **Sale** to true?*

By Expert, it should be the **Sale** itself, since it owns and maintains the isComplete attribute.

Register will send a ***becomeComplete*** message to the **Sale** to set it to true

Completion of item entry

Calculating the Sale Total

Consider this fragment of the Process Sale use case:

Main Success Scenario:

1. Customer arrives …
2. Cashier tells System to create a new sale.
3. Cashier enters item identifier.
4. System records sale line item and …
5. *Cashier repeats steps 3-4 until indicates done.*
6. System presents total with taxes calculated.

Calculating the Sale Total

In step 6, total is displayed

Due to Model-View Separation principle, display is a UI issue, not our concern for now,

But total must be known

Sale is an Expert and responsible for knowing its total

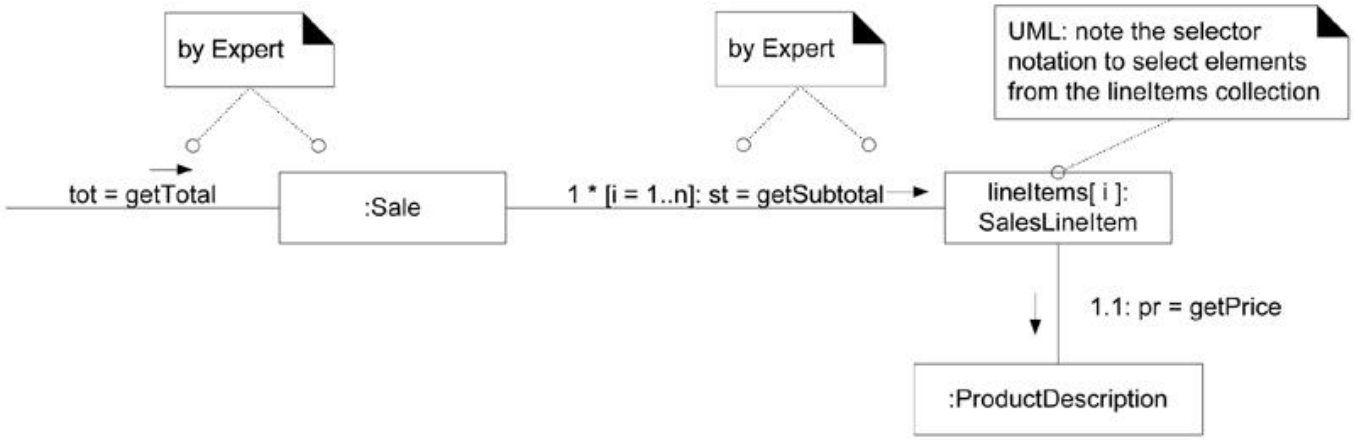
Calculating the Sale Total

1. State the responsibility:
    - *Who should be responsible for knowing the sale total?*
2. Summarize the information required:
    - The sale total is the sum of the subtotals of all the sales line-items.
    - sales line-item subtotal := line-item quantity * product description price
3. List the information required to fulfill this responsibility and the classes that know this information

Calculating the Sale Total

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

The Sale.*getTotal* Design



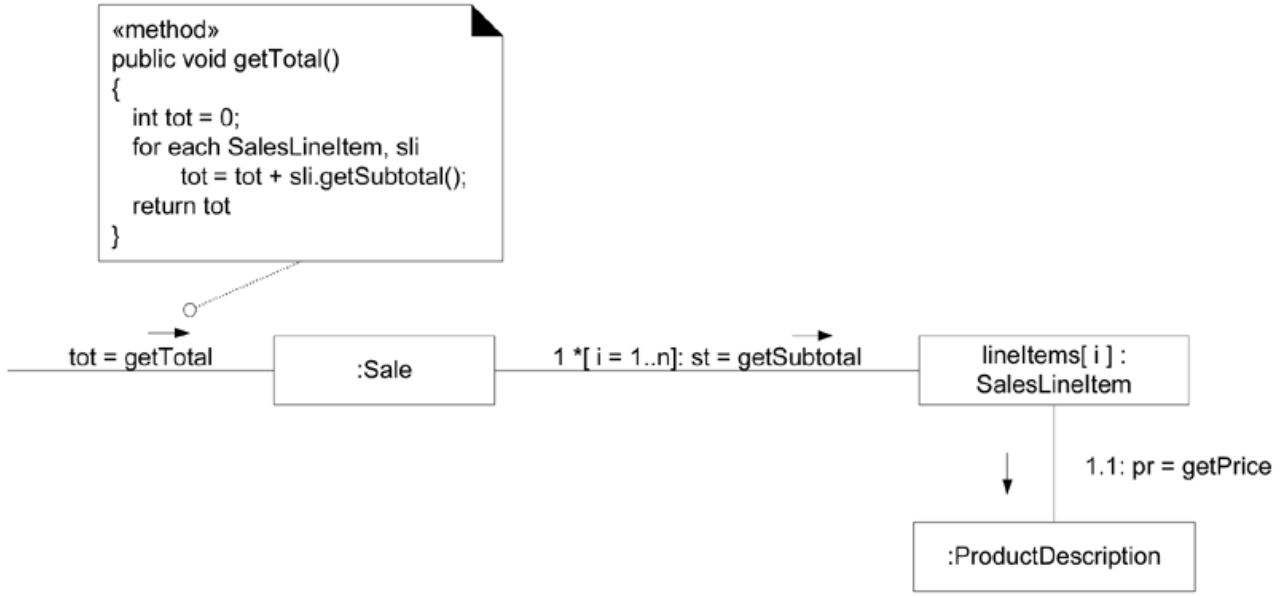
The Sale.*getTotal* Design

Since arithmetic is not (usually) illustrated via messages, we can illustrate the details of the calculations by attaching algorithms or constraints to the diagram that defines the calculations

*Who will send the **getTotal** message to the **Sale**?*

Most likely, it will be an object in the UI layer, such as a Java JFrame

The Sale.*getTotal* Design



How to Design *makePayment*?

Contract CO4: makePayment	
Operation	makePayment(amount: Money)
Cross References	Use Cases: Process Sale
Preconditions	There is a sale underway.
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)

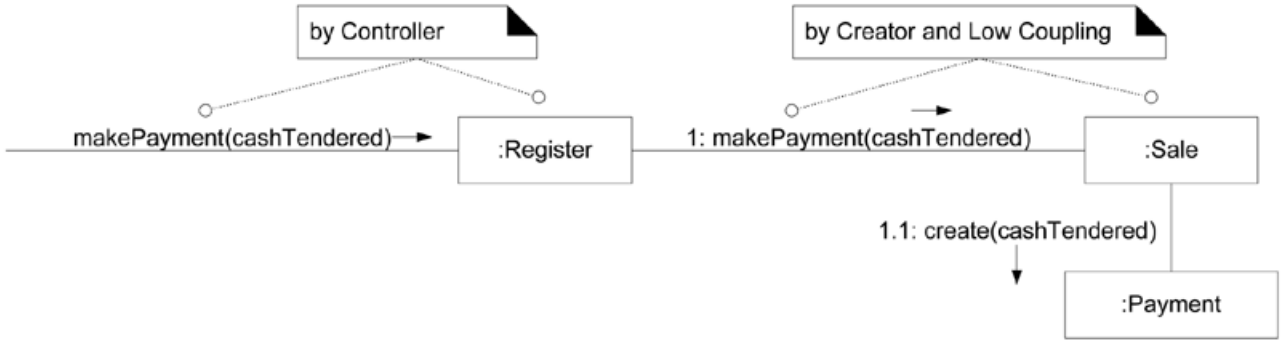
Creating the Payment

A **Payment** instance p was created (instance creation). This is a creation responsibility.

*Who records, aggregates, most closely uses, or contains a **Payment**?*

Sale software will closely use a **Payment**; thus, it, too, may be a candidate

Creating the Payment



This interaction diagram satisfies the postconditions of the contract:
the **Payment** has been created,
associated with the **Sale**,
and its amountTendered has been set.

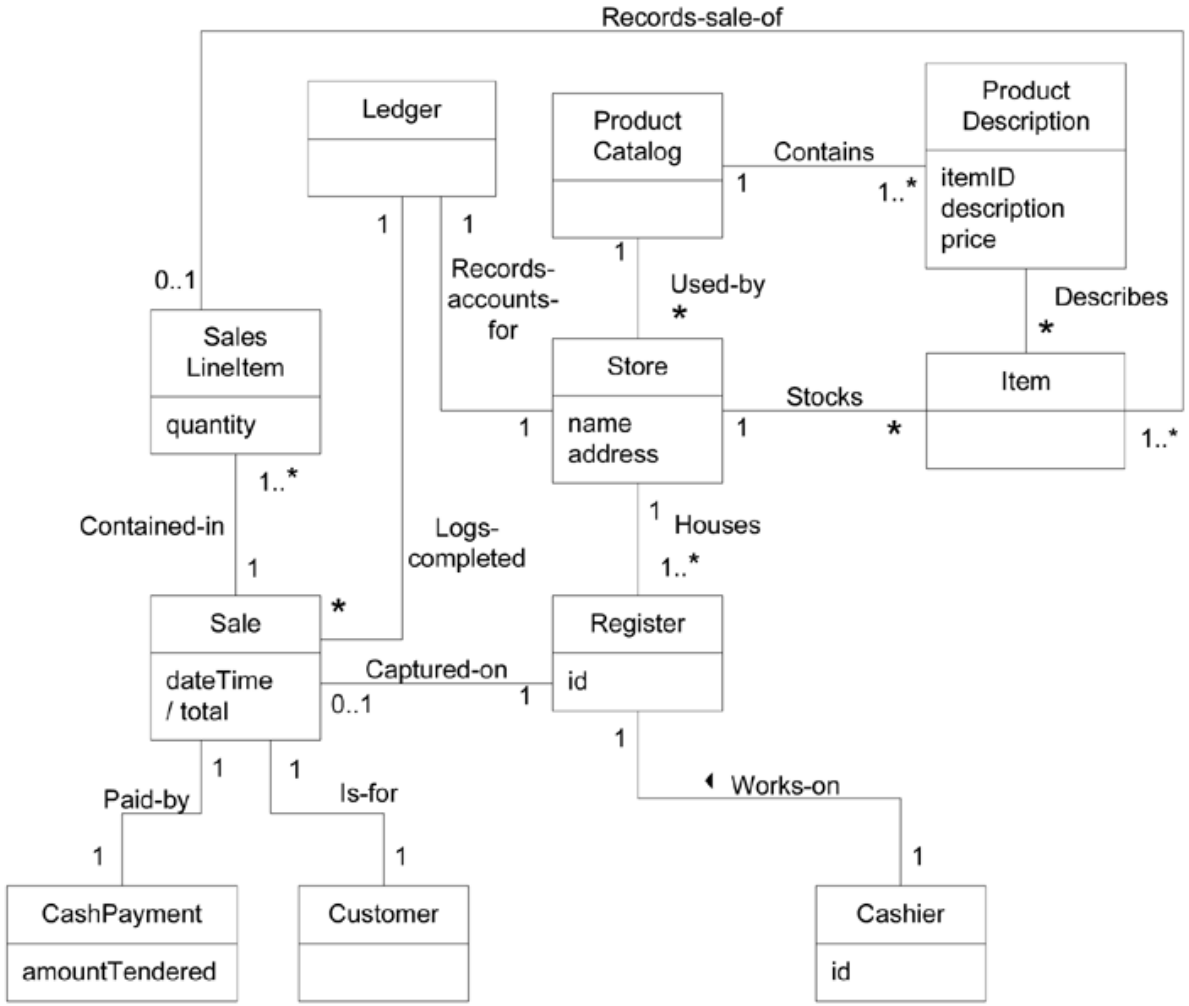
Logging a Sale

Once complete, the requirements state that the sale should be placed in an historical log

Information Expert should be an early pattern considered unless (Controller or Creation)

Who is responsible for knowing all the logged sales and doing the logging?

POS Partial Domain Model



Logging a Sale —A Conflict?

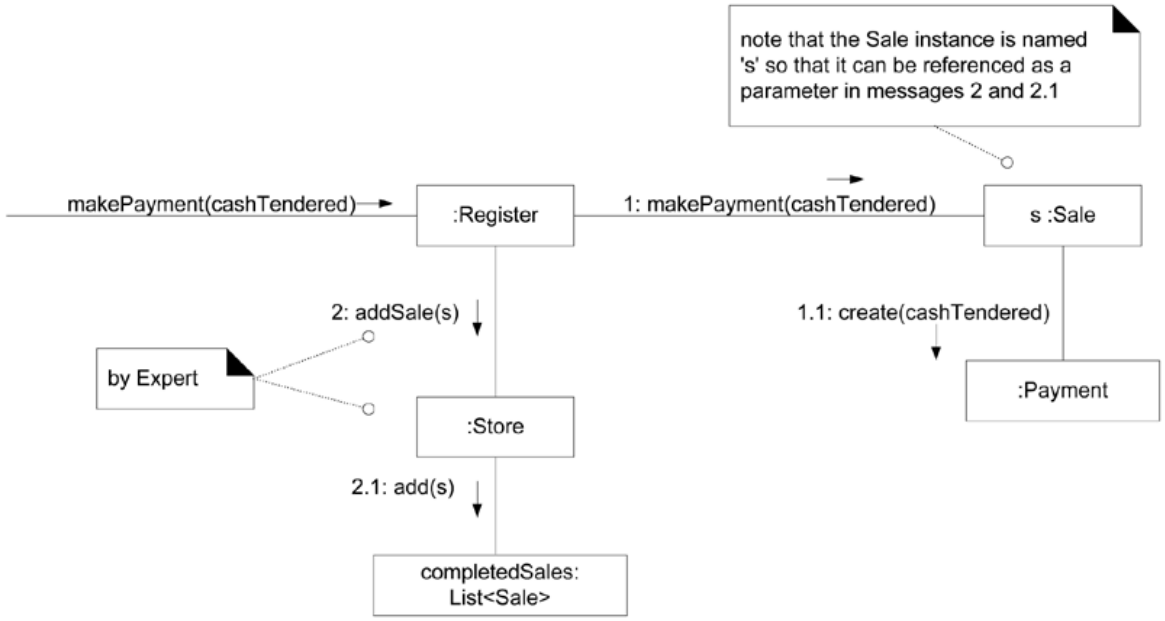
LRG in the software design, **Store** is a good candidate to know all the logged sales, **Store** interested in finances

- **SalesLedger** is a candidate. **Store** could grow and becomes incohesive

OR

- Postconditions of the contract relate **Sale** to the **Store**, we may stick with the original plan of using **Store**

Logging A Completed Sale



Calculating the Balance

The Process Sale use case implies that the balance due from a payment be printed on a receipt and displayed somehow

Because of the Model-View Separation principle, Not concern ourselves with how the balance will be displayed or printed, but we must ensure that it is known

No class currently knows the balance, **Information Expert** should be considered

*Who is responsible for **knowing** the balance?*

Calculating the Balance

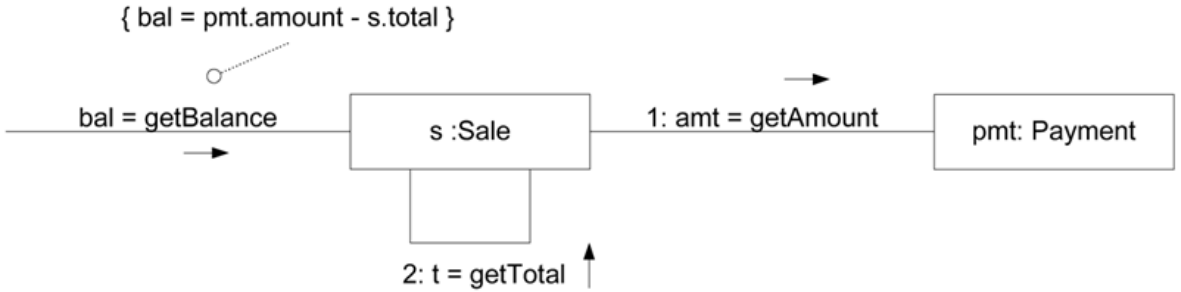
To calculate balance, we need the sale total and payment cash tendered, **Sale** and **Payment** are partial Experts on solving this problem

- If **Payment** is primarily responsible for **knowing** the balance, it needs visibility to the **Sale**, to ask **Sale** for its total
  - Since it does not currently know about the **Sale**, this approach would increase the overall coupling in the design—it would not support the Low Coupling pattern

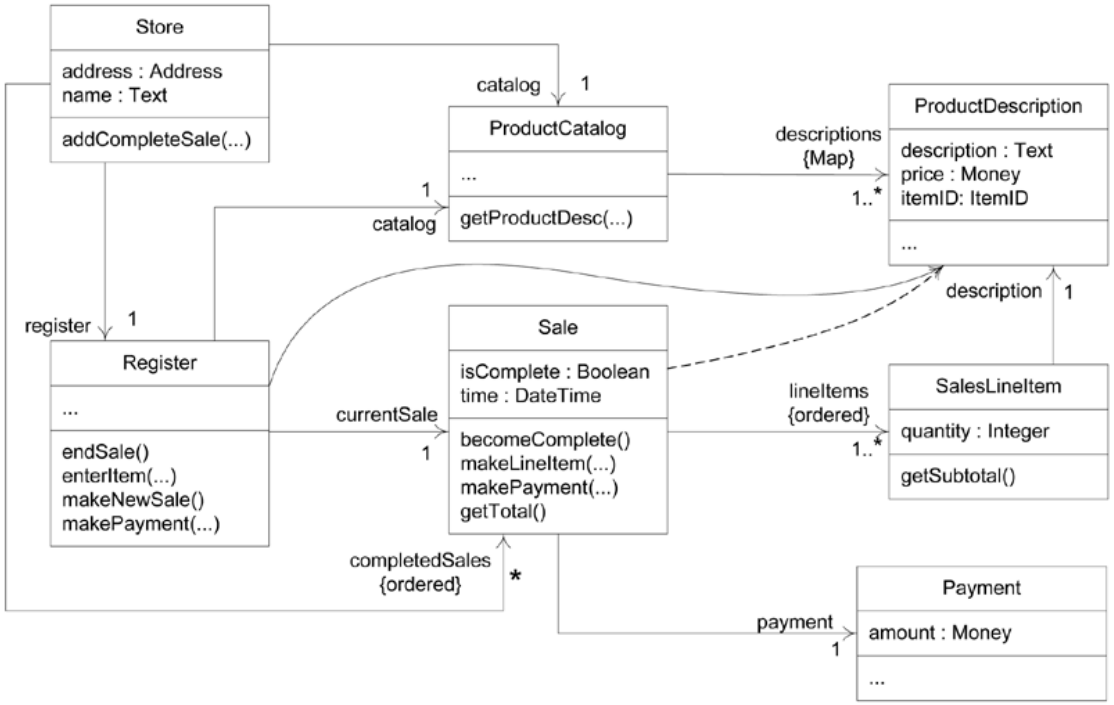
OR

- If **Sale** is primarily responsible for knowing the balance, it needs visibility to the **Payment** (how?), to ask it for its cash tendered
  - Since the **Sale** already has **visibility** to the **Payment**—as its creator—this approach does not increase the overall coupling and is therefore a preferable design

Sale.*getBalance* interaction diagram



Final POS DCD for Iteration-1



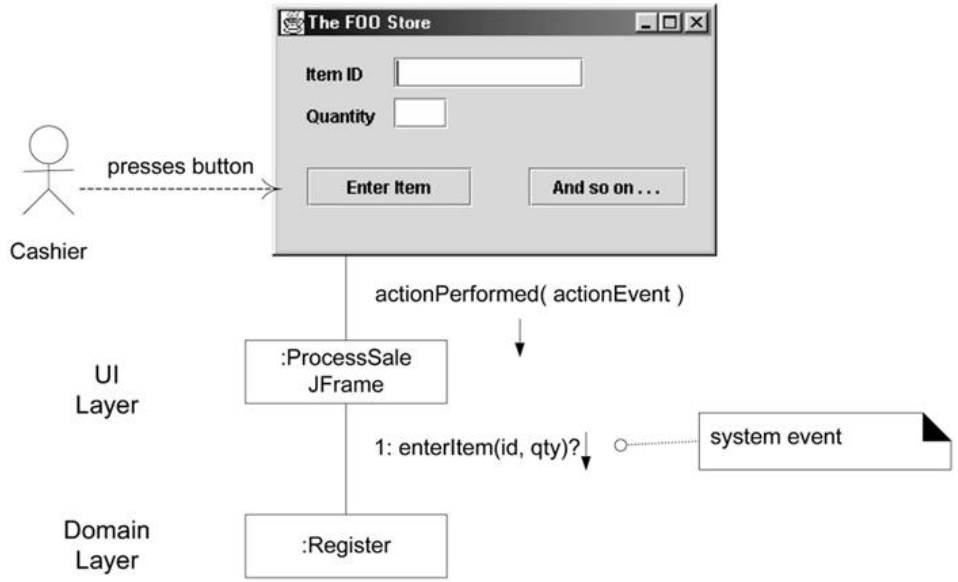
How to Connect the UI Layer to the Domain Layer?

Common designs by which objects in the UI layer obtain visibility to objects in the domain layer include the following:

- An initializer object called from the application starting method (e.g., ***main*** method)
- creates both a UI and a domain object and passes the domain object to the UI

- Once the UI object has a connection to the **Register** instance (**facade controller**), it can forward system event messages, such as the ***enterItem*** and ***endSale*** message, etc. to it

How to Connect the UI Layer to the Domain Layer?



How to Connect the UI Layer to the Domain Layer?

In the case of the ***enterItem*** message,

we want the window to show the **running total** after each entry.

Design solutions are:

Add a ***getTotal*** method to the **Register**.

- UI sends ***getTotal*** message to **Register**, which delegates to **Sale**. And maintaining lower coupling from UI to domain layer— UI only knows of **Register**,
- But it starts to expand the interface of the **Register**, making it less cohesive

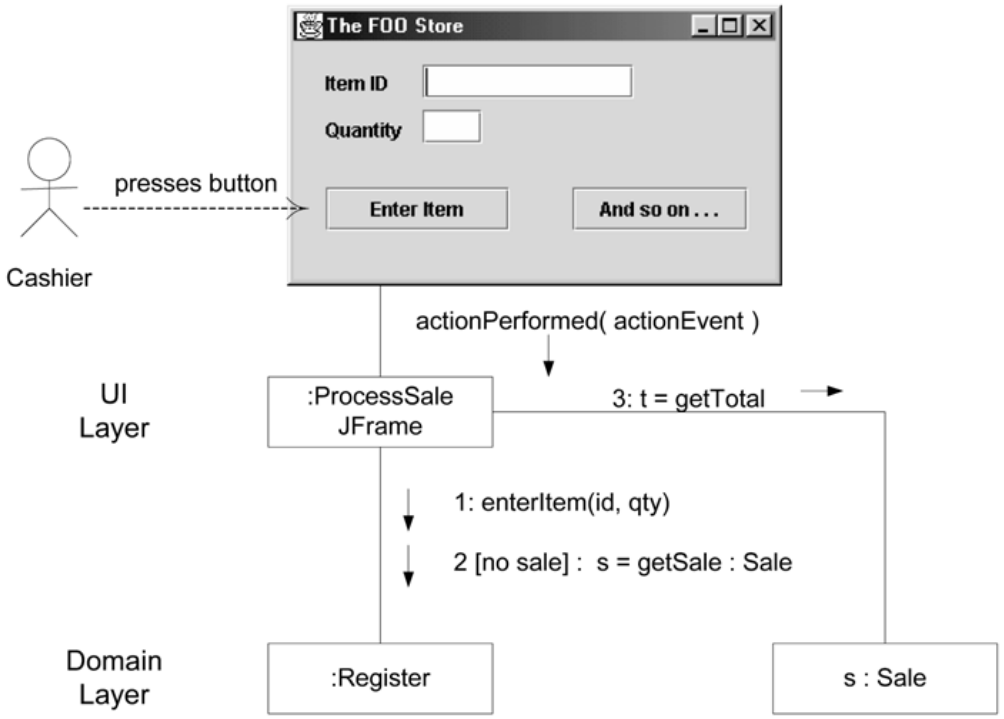
How to Connect the UI Layer to the Domain Layer?

OR

UI gets a reference to **Sale**, and then ***getTotal*** (or any other information related to the sale)

- UI directly sends messages to the **Sale**
- This design increases coupling UI to domain layer
- Sometimes higher coupling is not the problem; rather, coupling to unstable objects is a real problem
- If **Sale** is a stable object that will be an integral part of the design—which is reasonable. Coupling to the Sale is not a major problem

Connecting The UI And Domain Layers



When to Create the Initialization Design?

Systems have either an **implicit** or **explicit** Start Up use case

A StartUp system operation is the earliest one to execute

Delay the development of an interaction diagram for it until after all other system operations have been considered

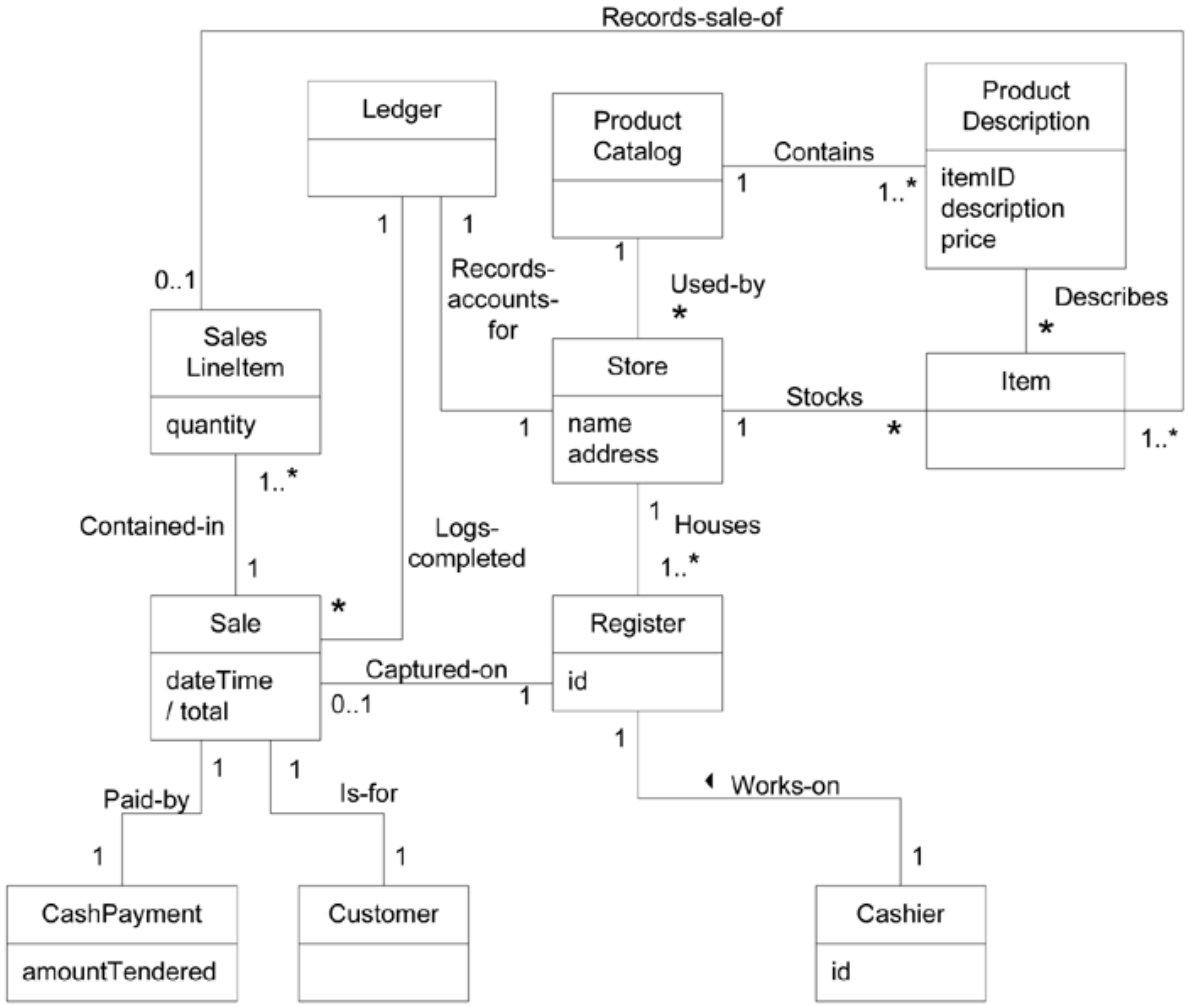
How do Applications Start Up?

This creation may happen explicitly in the starting main method or delegated to a Factory object called from the main method

Often, the initial domain object, once created, is responsible for the creation of its direct child domain objects

- For example, a **Store** chosen as the **initial** domain object may be responsible for the creation of a **Register** object

POS Partial Domain Model



Choosing the Initial Domain Object

Choose as an initial domain object a class at or near the **root** of the aggregation hierarchy of domain objects

A facade controller, such as **Register**, or some other object considered to contain all or most other objects, such as a **Store**

High Cohesion and Low Coupling considerations influence the choice between these alternatives. In this application, we chose the **Store** as the initial object

How do Applications Start Up?

```
public class Main{
        public static void main( String[] args ){
          // Store is the initial domain object.
          // The Store creates some other domain objects.
          Store store = new Store();
          Register register = store.getRegister();
          ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
          ...
        }
}
```

Store.Create Design

By reflecting on the prior interaction designs, we identify the following initialization work:

Create a **Store**, **Register**, **ProductCatalog**, and **ProductDescriptions**

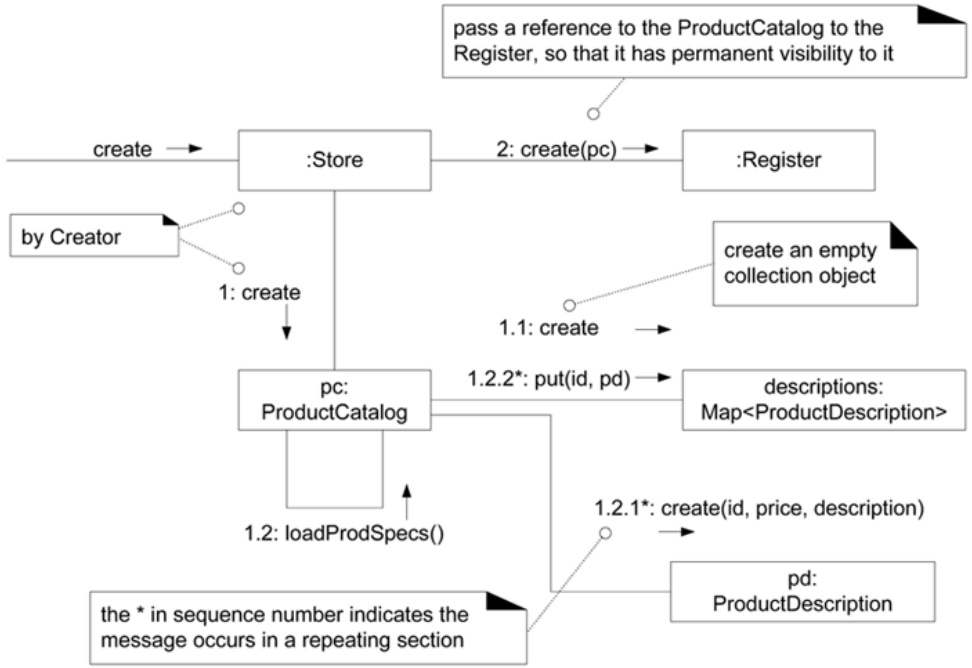
Associate the **ProductCatalog** with **ProductDescriptions**

Associate **Store** with **ProductCatalog**

Associate **Store** with **Register**

Associate **Register** with **ProductCatalog**

Creation of Initial Domain Object And Subsequent Objects



Applying UML

Observe that the creation of all the **ProductDescription** instances and their addition to a container happens in a repeating section, indicated by the * following the sequence numbers

Multiplicity between classes of objects in the Domain Model and Design Model may not be the same