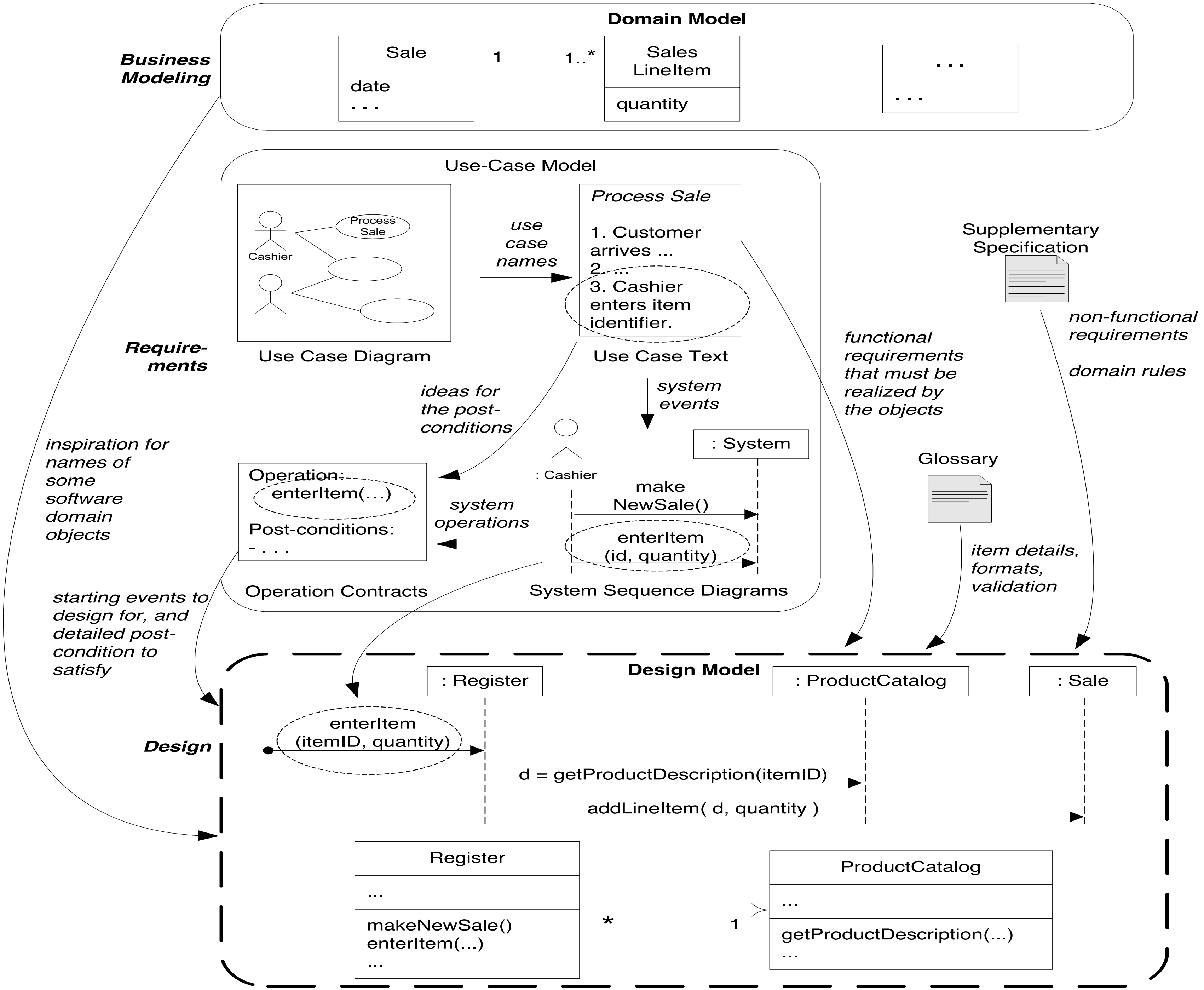
Chapter 18

Object Design Examples with GRASP

Objectives

- Design use case realizations
  - A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects [RUP]
- Apply GRASP to assign responsibilities to classes
- Apply UML to illustrate and think through the design of objects
- In this chapter, we will apply OO design principles and the UML to the case studies “POS system and Monopoly”.

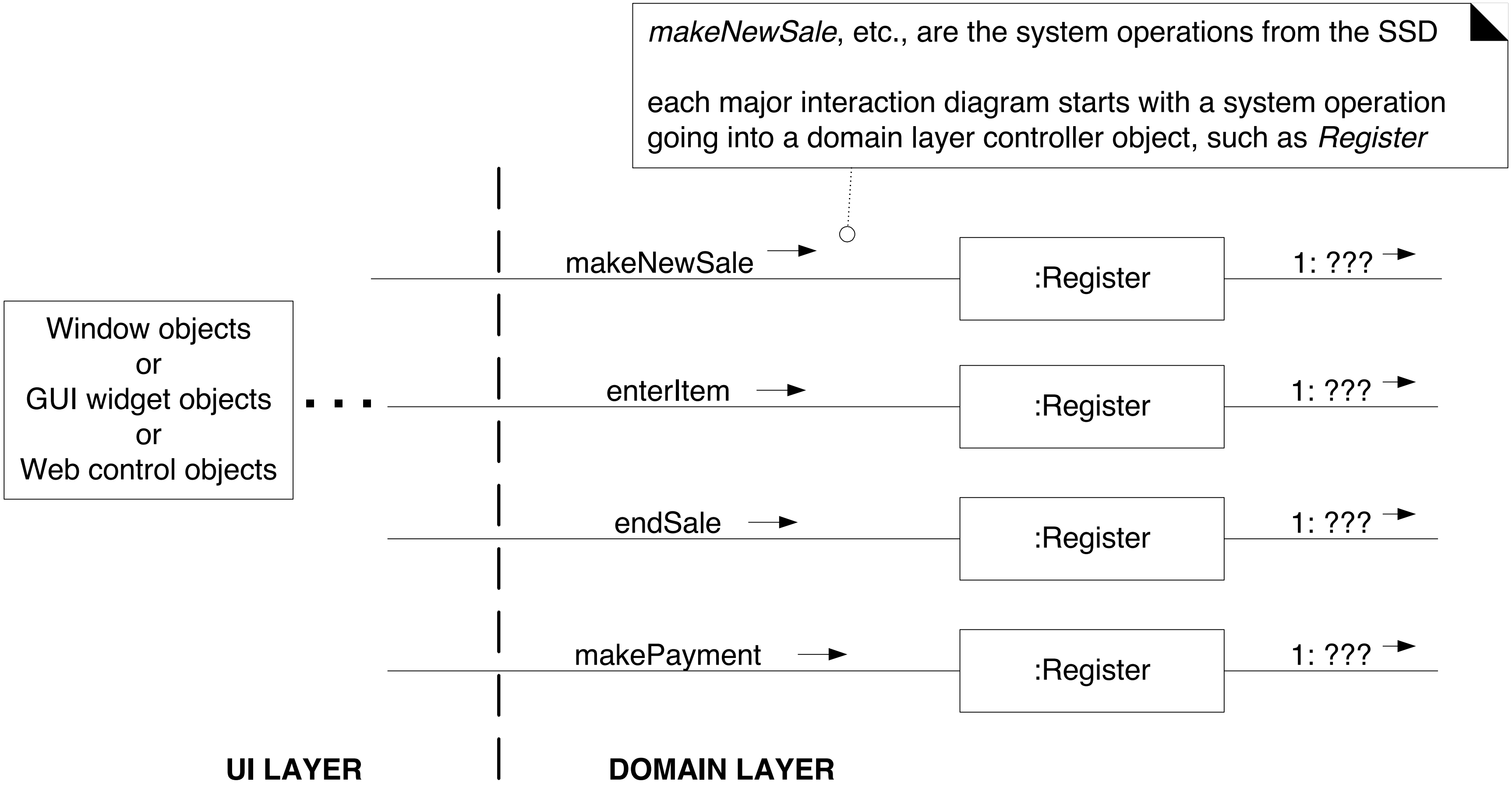
Sample UP Artifact Relationships



Use case realization

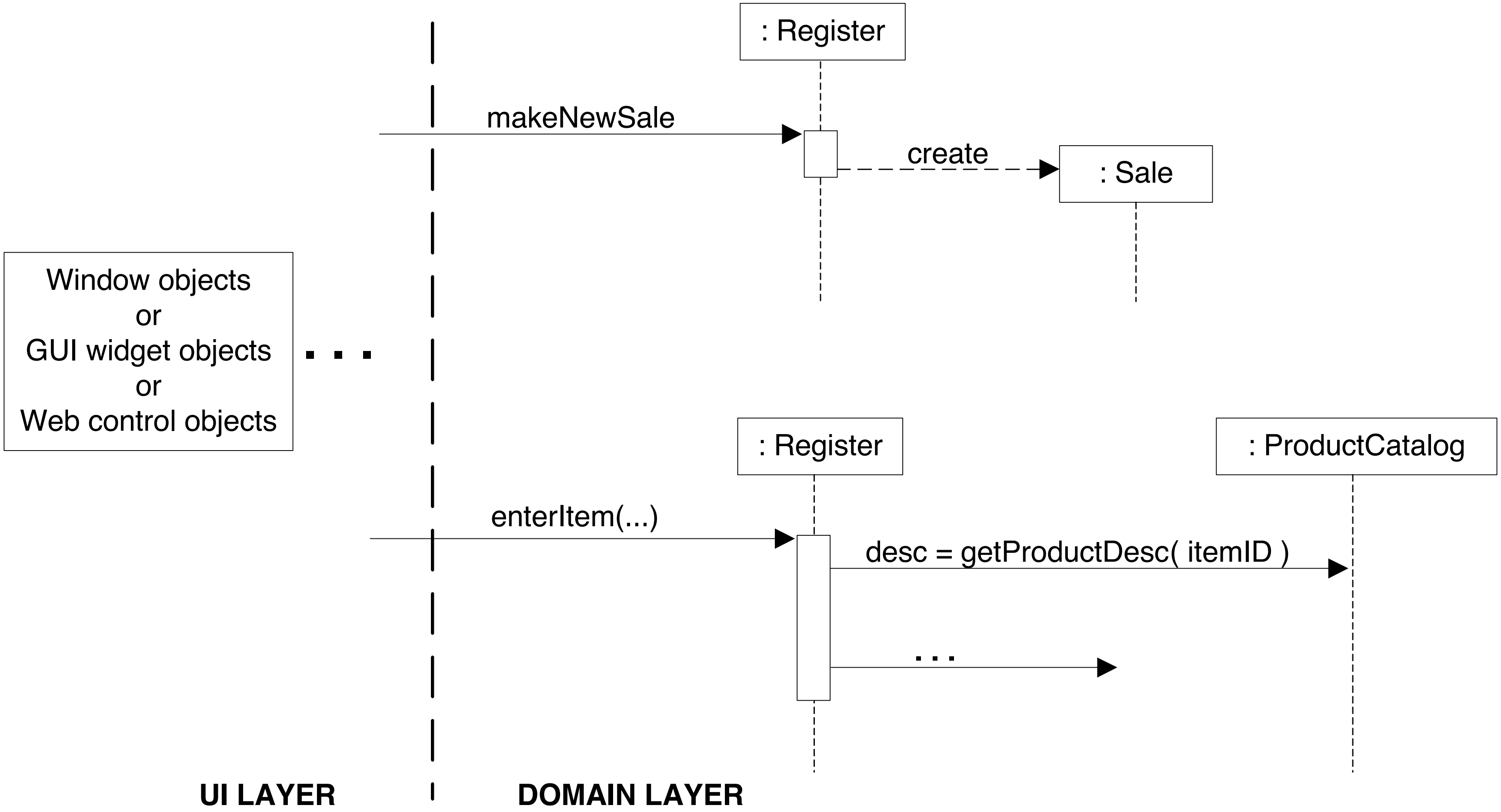
- A use-case realization describes how a particular use case is realized within the design model, in terms of collaborating objects [RUP]
- UML diagrams are a common language to illustrate use case realizations
- Relationship between some UP artifacts emphasizing use case realizations:
  - Use case suggests the system operations that are shown in SSDs
  - System operations become starting messages entering the Controllers for domain layer interaction diagrams
  - Domain layer interaction diagrams illustrate how objects interact to fulfill the required tasks – **the use case realization**.

Fig. 18.2


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
Window objects or GUI widget objects or Web control objects
makeNewSale
:Register
1: ???
enterItem
:Register
1: ???
endSale
:Register
1: ???
makePayment
:Register
1: ???
UI LAYER
DOMAIN LAYER

Scenarios and systems operations identified on SSDs of the Process Sale Use case

Fig. 18.3



Sequence Diagrams and System Operation Handling

Use case realization - cont.

- Uses cases are prime input to use case realization
- Related documents
  - Supplementary Specifications
  - Glossary
  - UI prototypes
  - ….
- All inform developers what needs to be built
- Bear in mind: written requirements are imperfect
  - Involve the costumer frequently
  - Evaluate demos, discuss requirements, tests etc
- For complex system operations:
  - Operation contracts may have been written
  - Work through post-condition state changes and design message interactions to satisfy requirements
- Domain model inspires some software objects
  - It is normal to discover new concepts during design that were missed earlier in domain analysis, or ignore concepts that were previously identified

Case study: Process Sale

Use Case Realization for NextGen Iteration

- Choices and decisions involved in the design of use case realizations:
  - makeNewSale
  - enterItem
  - endSale
  - makePayment
- Based on GRASP patterns
- Final design NextGen DCD
- How to connect UI Layer to Domain Layer
- How do applications startup

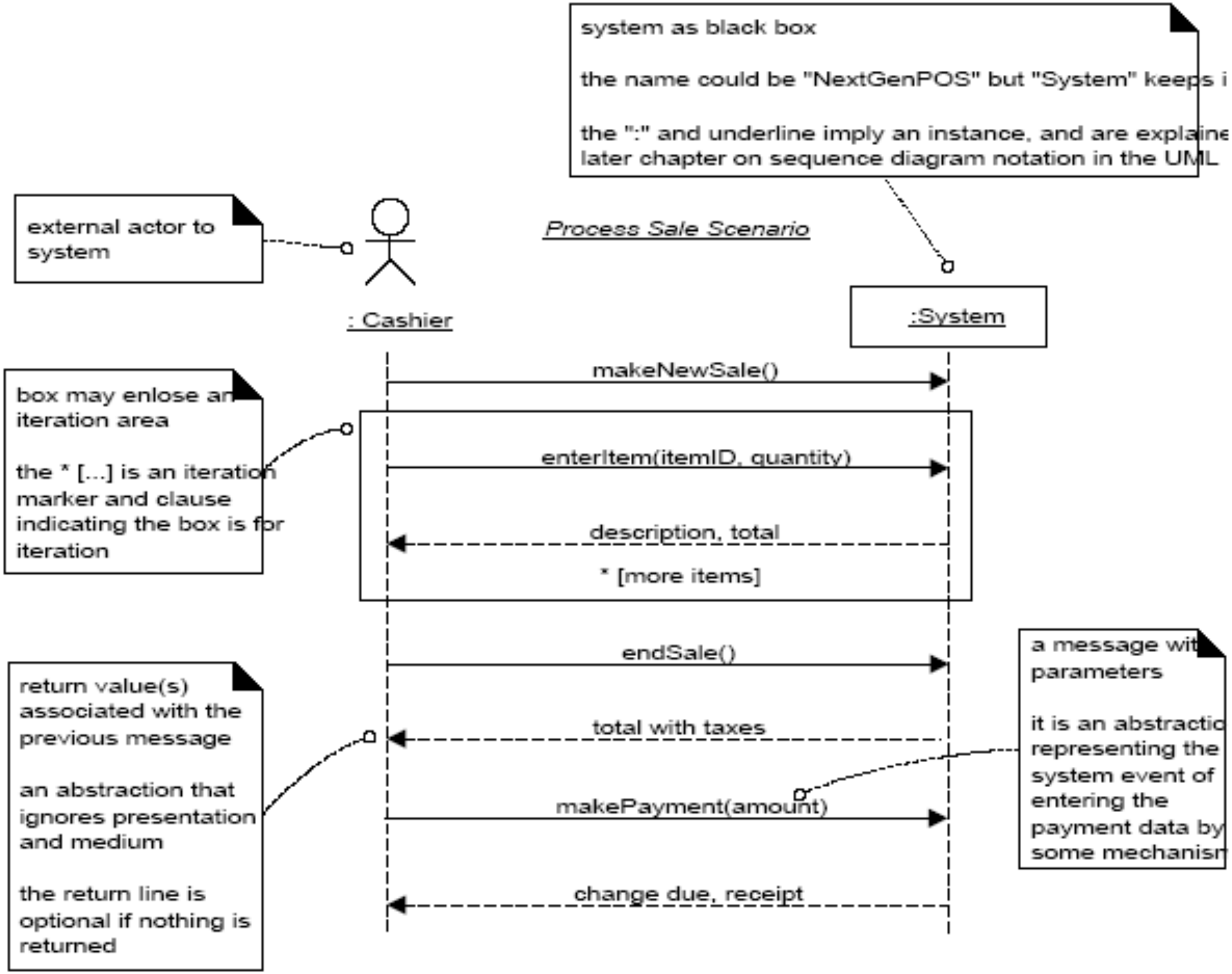
Use Case UC1: Process Sale

Primary Actor: ...

... as before ...

Main Success Scenario:

Actor Action (or Intention)	System Responsibility
1. Customer arrives at a POS checkout with goods and/or services to purchase.	
2. Cashier starts a new sale.	
3. Cashier enters item identifier.	4. Records each sale line item and presents item description and running total.
Cashier repeats steps 3-4 until indicates done.	5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.	
7. Customer pays.	8. Handles payment.



system as black box

the name could be "NextGenPOS" but "System" keeps i

the ":" and underline imply an instance, and are explaine
later chapter on sequence diagram notation in the UML

external actor to
system

box may enlose an
iteration area

the * [...] is an iteration
marker and clause
indicating the box is for
iteration

a message wit
parameters

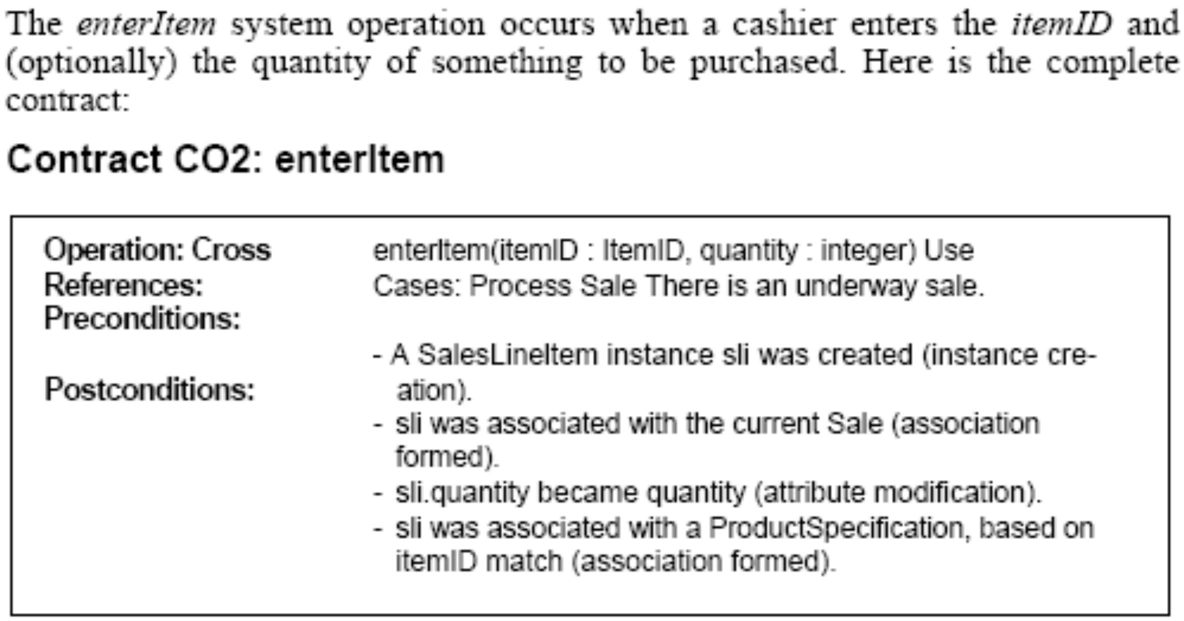
it is an abstractio
representing the
system event of
entering the
payment data by
some mechanism

return value(s)
associated with the
previous message

an abstraction that
ignores presentation
and medium

the return line is
optional if nothing is
returned

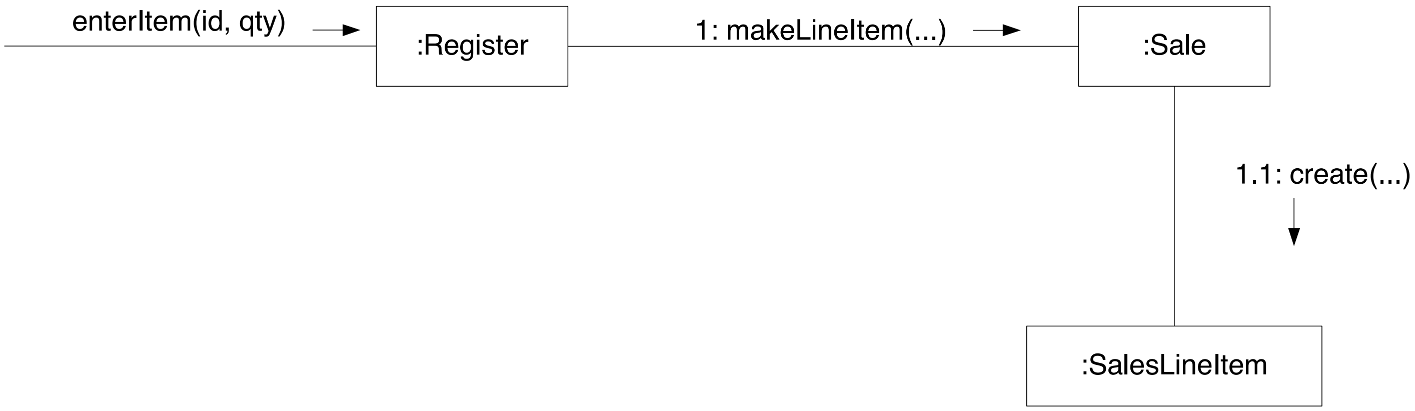
Operations Contracts and Use Case Realizations



The *enterItem* system operation occurs when a cashier enters the *itemID* and (optionally) the quantity of something to be purchased. Here is the complete contract:

Contract CO2: enterItem

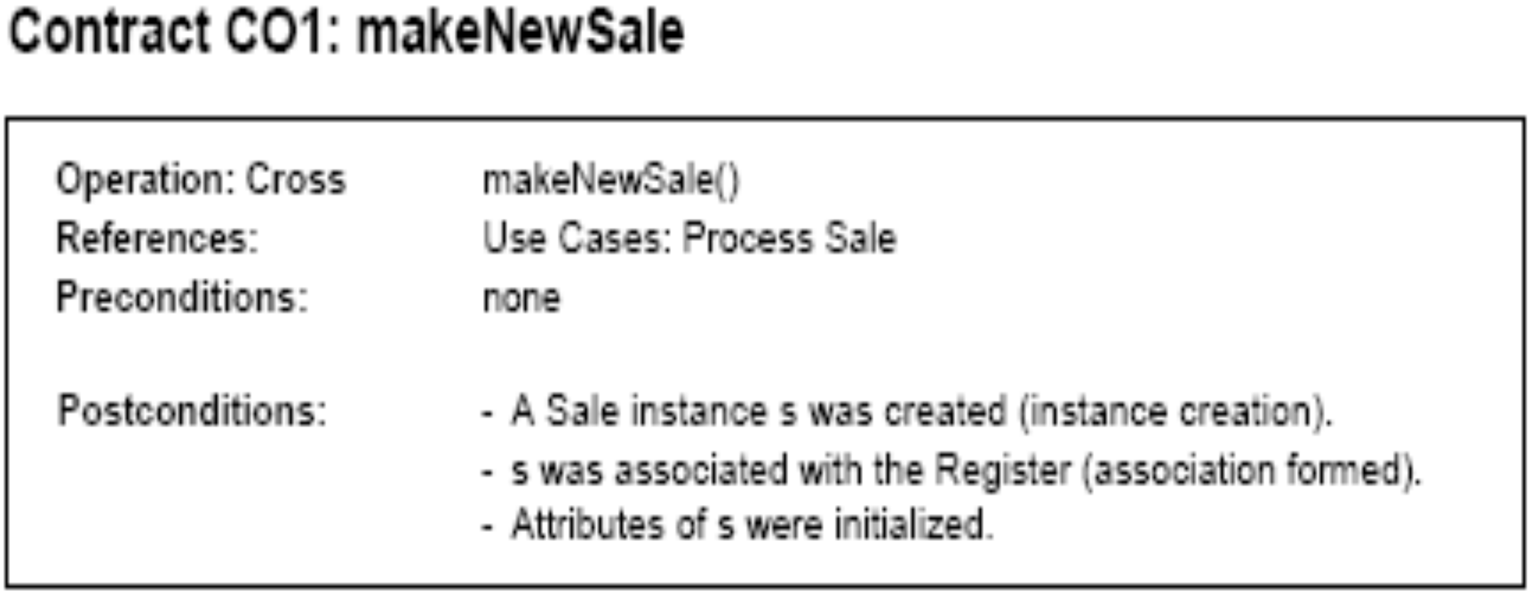
Operation: Cross	enterItem(itemID : ItemID, quantity : integer) Use
References:	Cases: Process Sale There is an underway sale.
Preconditions:	
Postconditions:	- A SalesLineItem instance sli was created (instance creation). - sli was associated with the current Sale (association formed). - sli.quantity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



USE CASE REALIZATIONS FOR NEXTGEN

- More detailed discussion - we will explore the choices and decisions made during design of a use case realization (Process Sale) with objects based on GRASP patterns.
- Initialization (or Start Up Use Case) Realization
  - Design context in which creation of most “root” or long-lived objects are considered
  - When coding, program at least some Start Up initialization first.
  - But during OOD design modeling, consider Start Up last, after you have discovered what needs to be created.
  - We will explore ProcessSale use case realization first, before Start Up

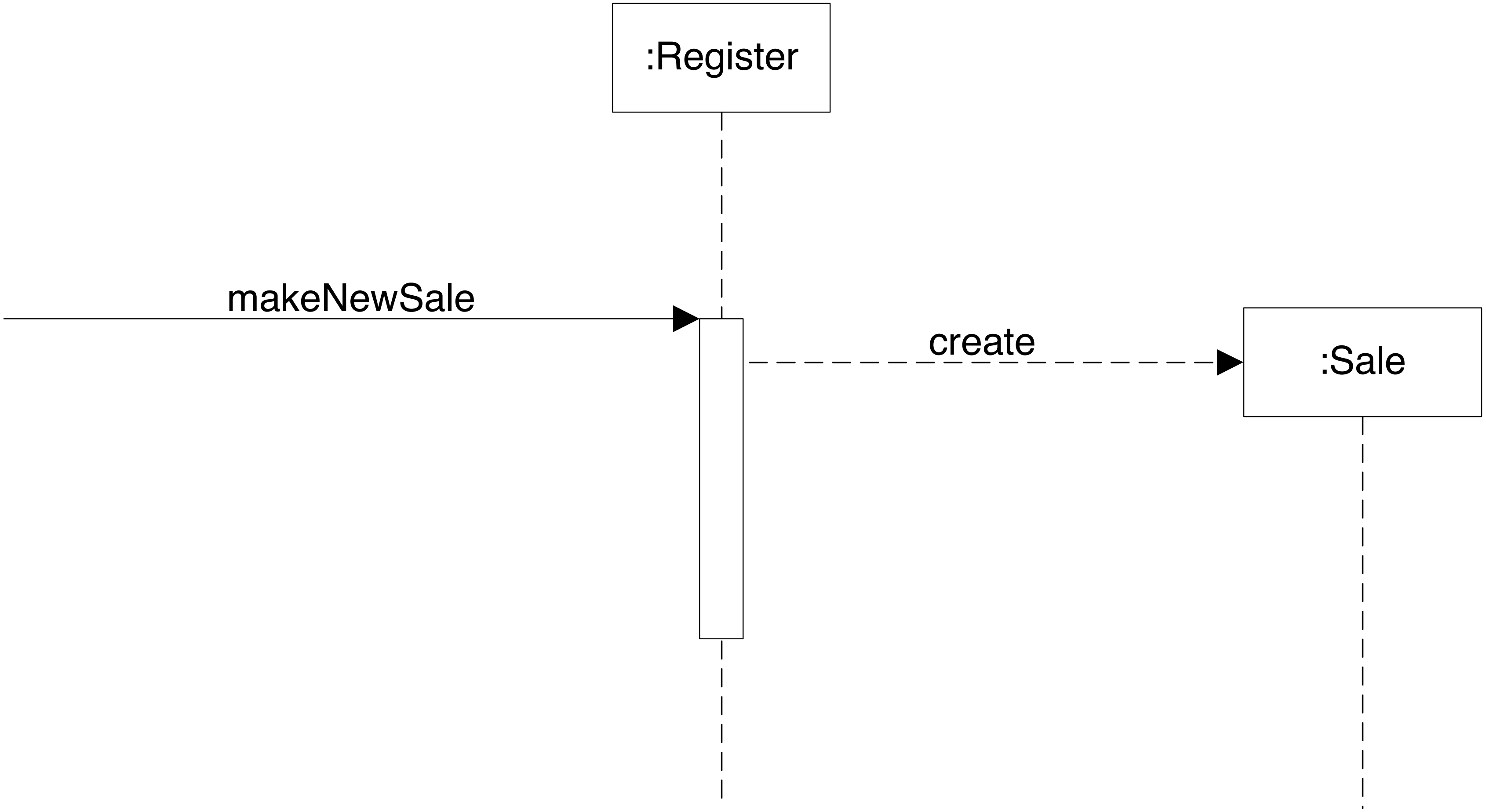
How to Design makeNewSale



Contract CO1: makeNewSale

Operation: Cross	makeNewSale()
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.

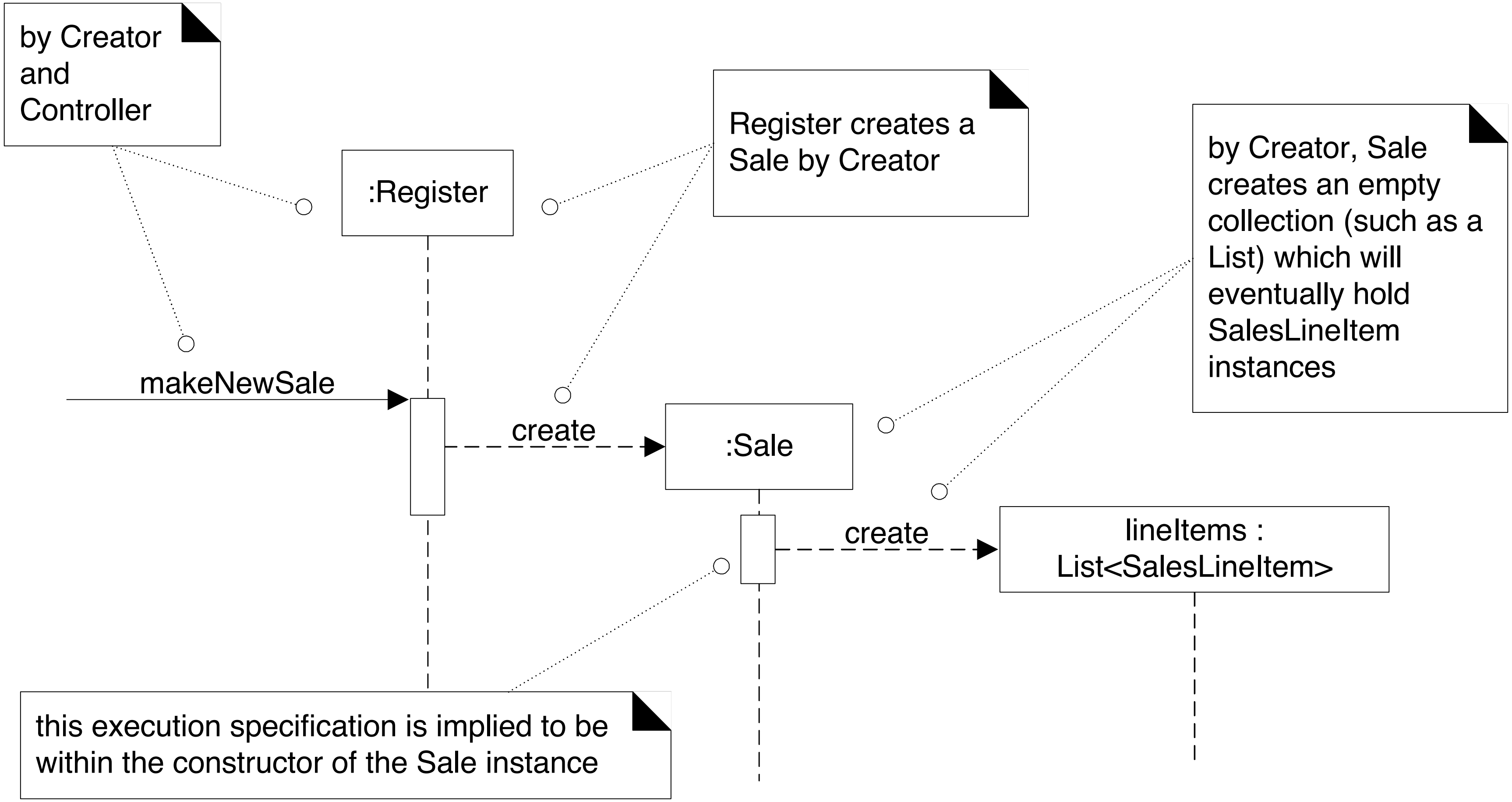
Choosing the Controller Class



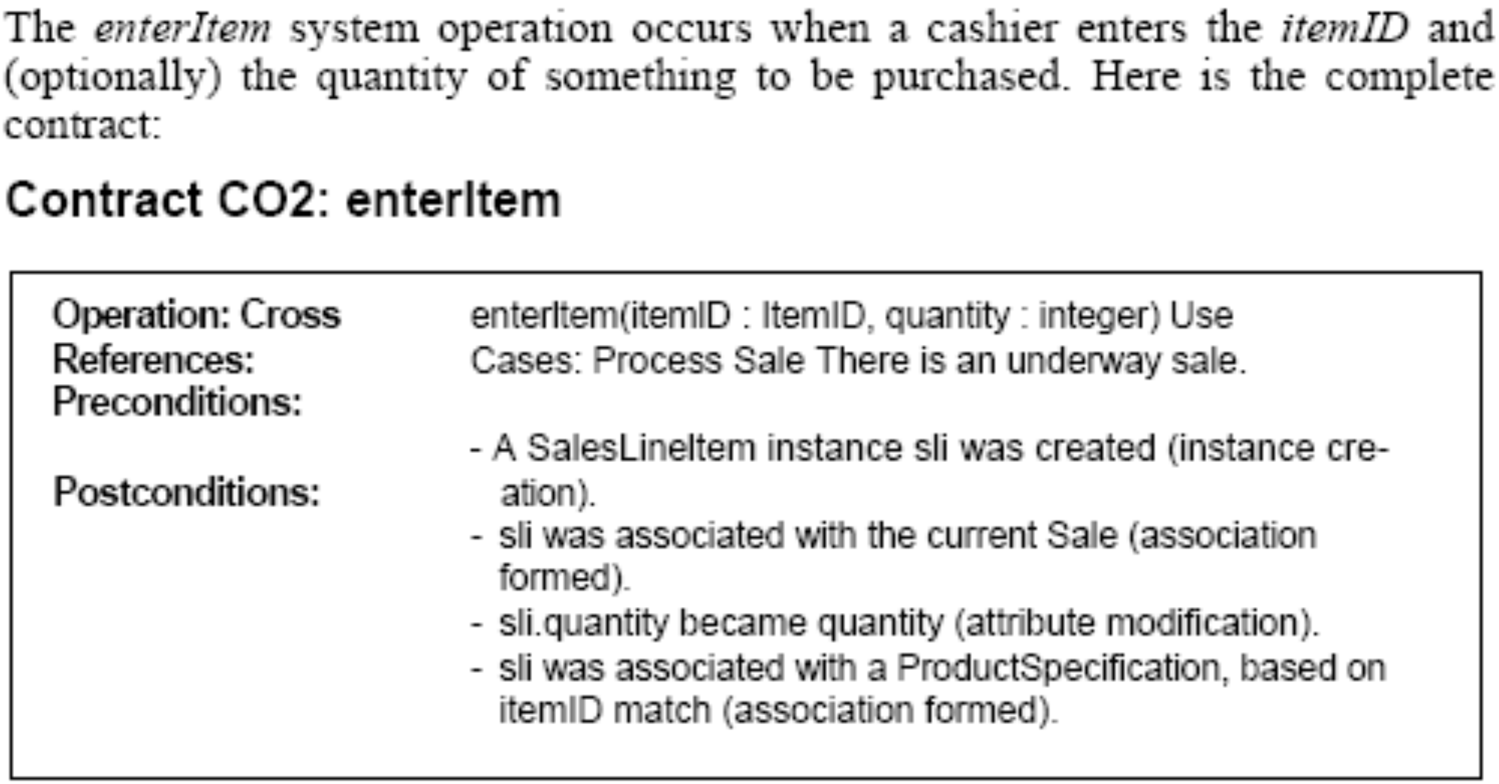
Applying the GRASP Controller Pattern

Creating New Sale

Fig. 18.6


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

How to Design enterItem



The *enterItem* system operation occurs when a cashier enters the *itemID* and (optionally) the quantity of something to be purchased. Here is the complete contract:

Contract CO2: enterItem

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

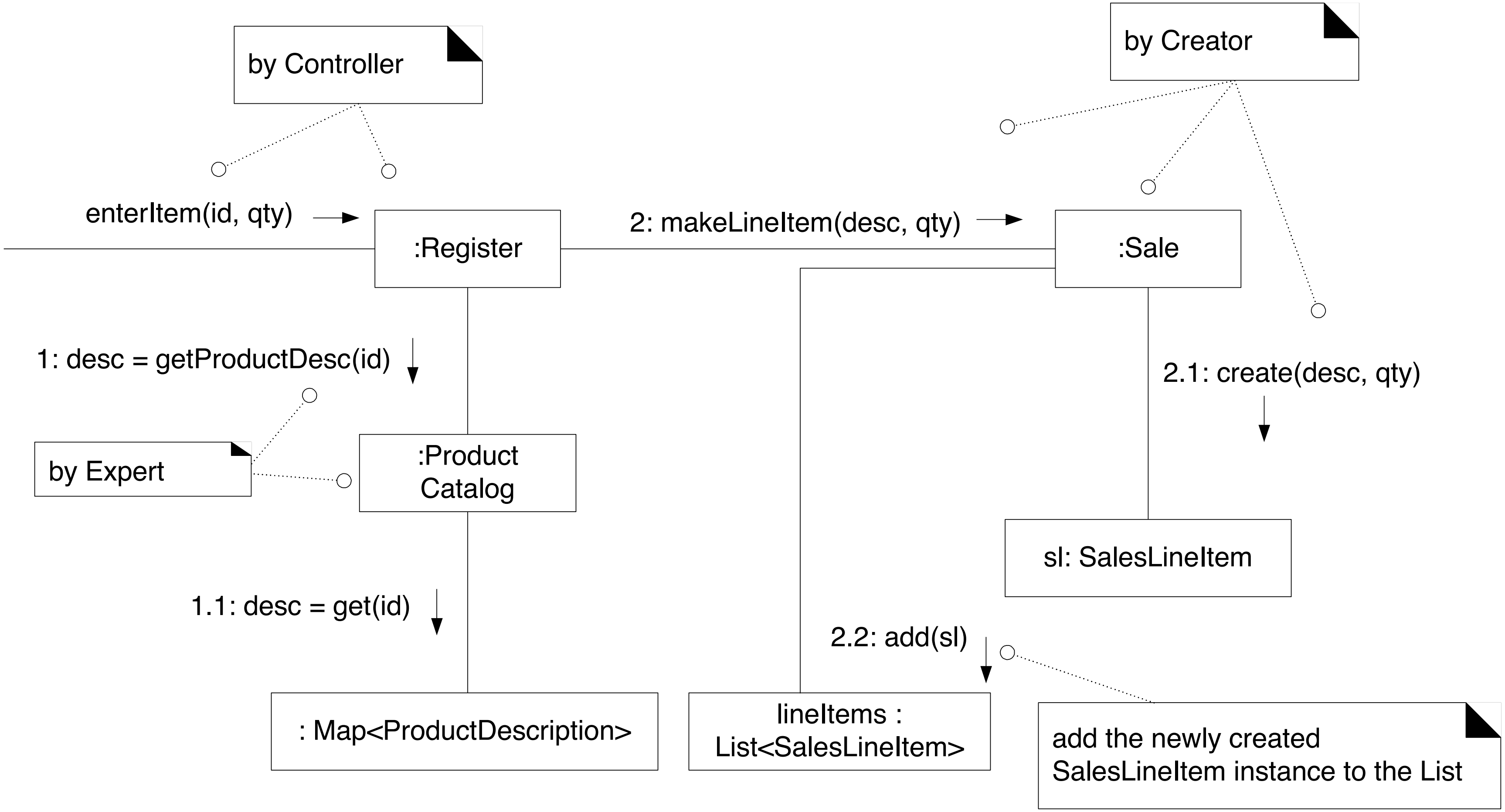
Constructing interaction diagram for enterItem using GRASP Patterns

- Choosing controller class
  - Will continue to use Register
- Display Item Description and Price
  - Use case states that output is displayed after operation
  - Model-View-Separation : it is not responsibility of non-GUI objects (Register, Sale) to get involved in output tasks.
  - We ignore it for now, but will handle it soon
  - All that is required with respect to the responsibilities for the display is that the information is known.
- Creating a new SalesLineItem
  - enterItem contract indicates creation, initialization, and association of SaleLineItem
  - Analysis of domain objects -> Sale contains SaleLineItems

Constructing interaction diagram for enterItem using GRASP Patterns

- Finding ProductDescription
  - SalesLineItem needs to be associated with ProductDescription that matches incoming itemID.
    - Who should be responsible for knowing a ProductDescription, based on itemID match
    - Information Expert Pattern
    - Analyzing domain model: ProductCatalog contains all ProductDescriptions
- Visibility to a ProductCatalog
  - Who should send the getProductDescription message to ProductCatalog to ask for a ProductDescription

Fig. 18.7 enterItem Interaction Diagram (Dynamic View)



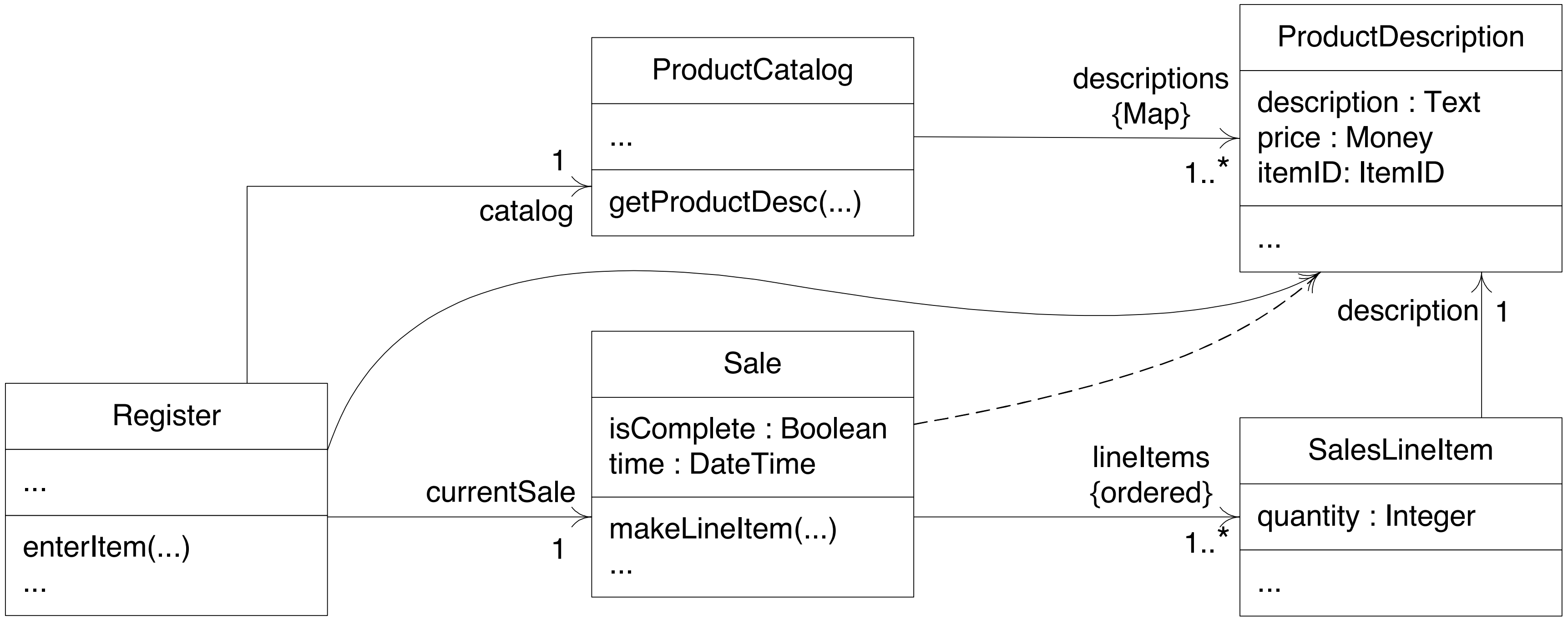
by Controller

by Creator

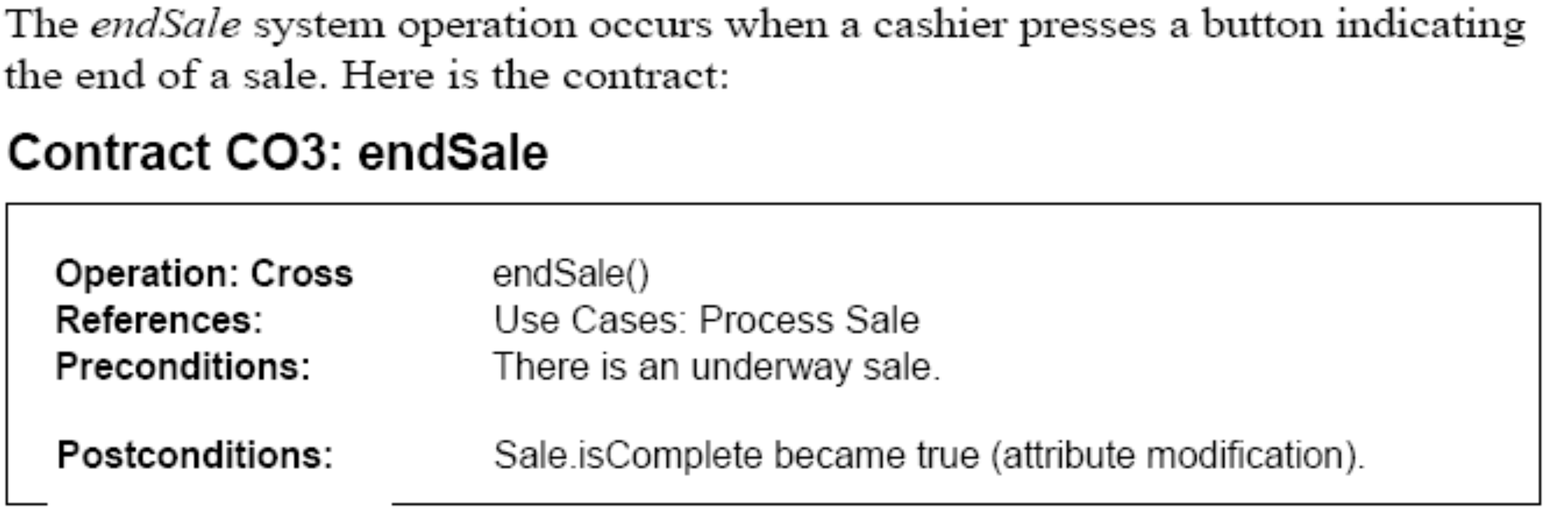
by Expert

add the newly created SalesLineItem instance to the List

Fig. 18.8 Partial DCD related to the enterItem (Static view)



How to design endSale



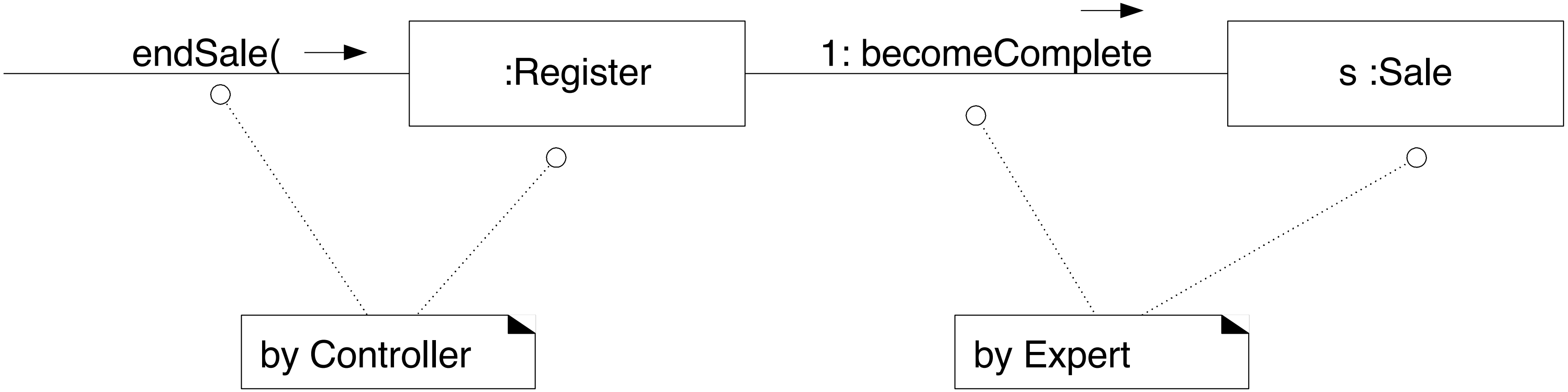
The *endSale* system operation occurs when a cashier presses a button indicating the end of a sale. Here is the contract:

Contract CO3: endSale

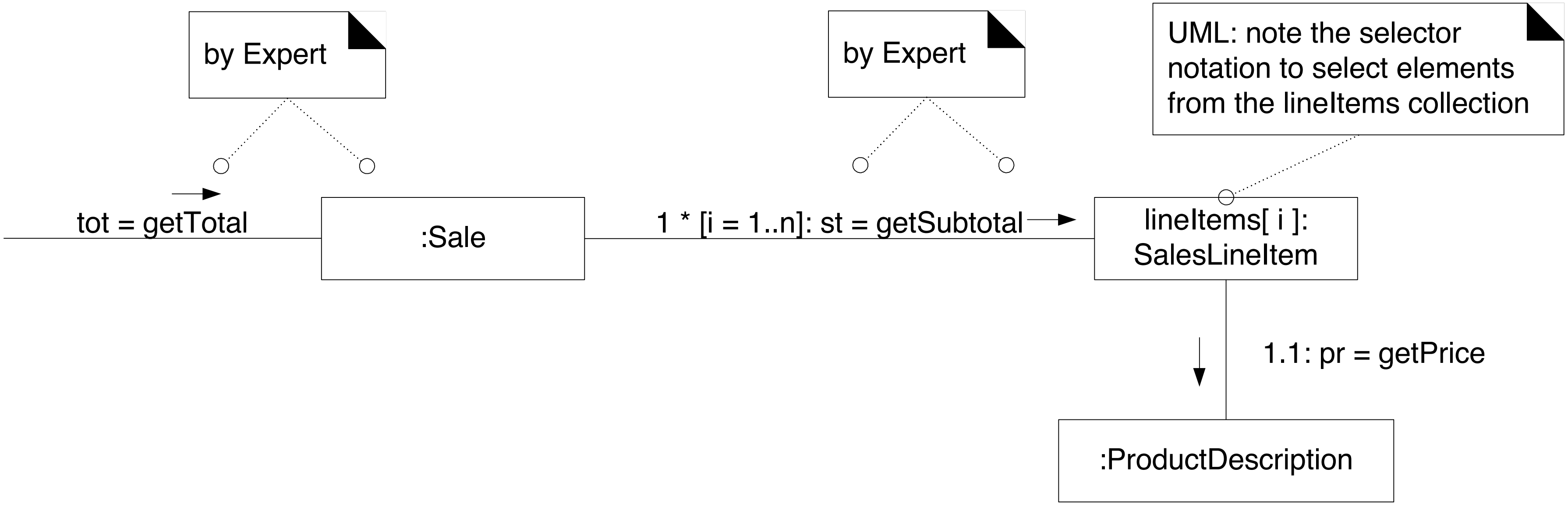
Operation: Cross	endSale()
References:	Use Cases: Process Sale
Preconditions:	There is an underway sale.
Postconditions:	Sale.isComplete became true (attribute modification).

How to design endSale

- Choosing controller: Register
- Setting Sale.isComplete attribute
  - By Expert, it should be Sale
  - Thus, Register will send a becomeComplete message to Sale



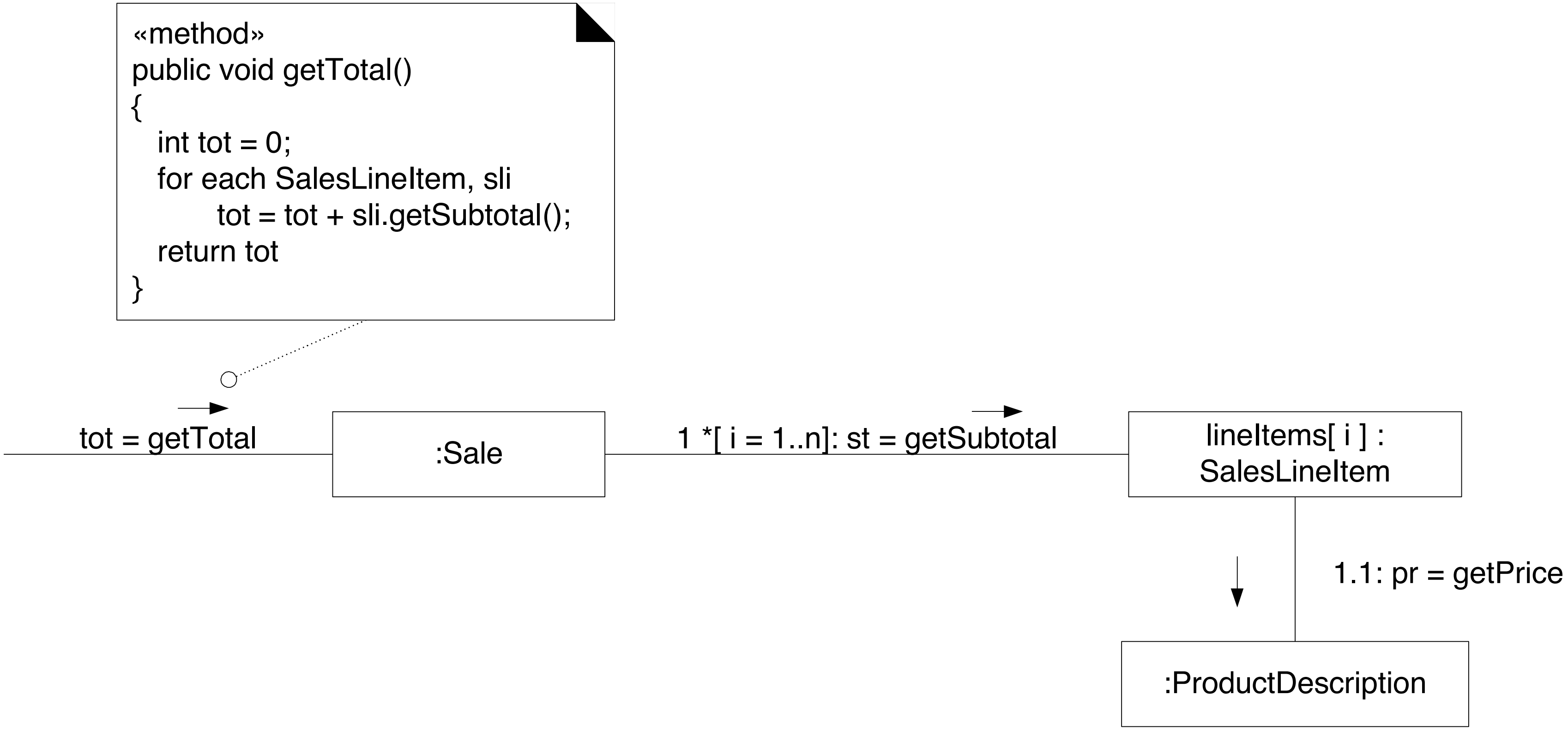
How to design endSale - calculating the sale total



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

- Not every interaction diagram starts with a system event message; they can start with any message for which the designer wishes to show interactions.
- 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. (will discuss in a minute).*

Showing a method in a note symbol



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

tot = getTotal

:Sale

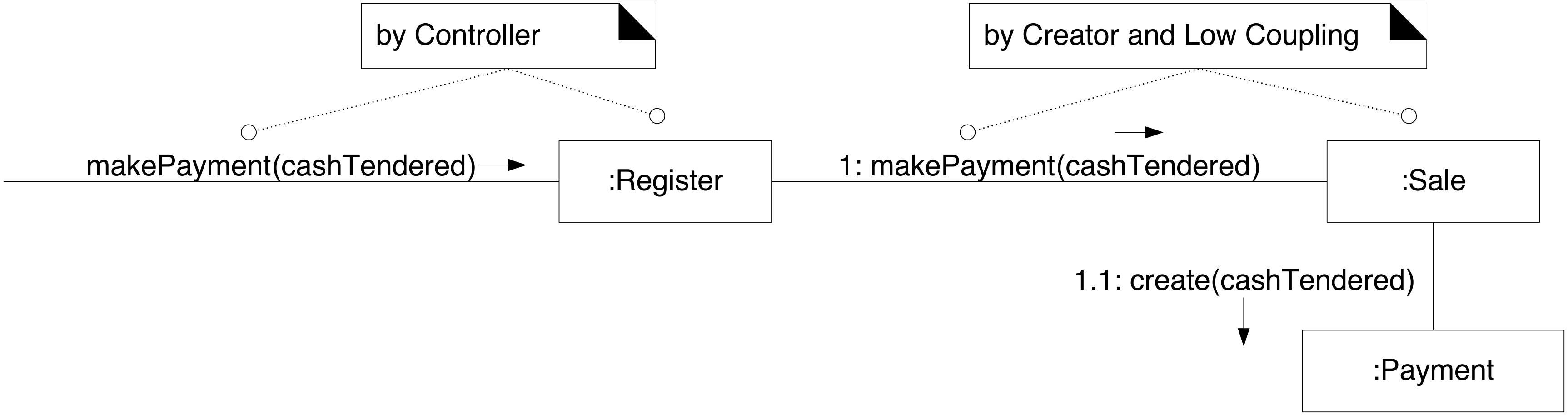
1 *[i = 1..n]: st = getSubtotal

lineItems[i] :
SalesLineItem

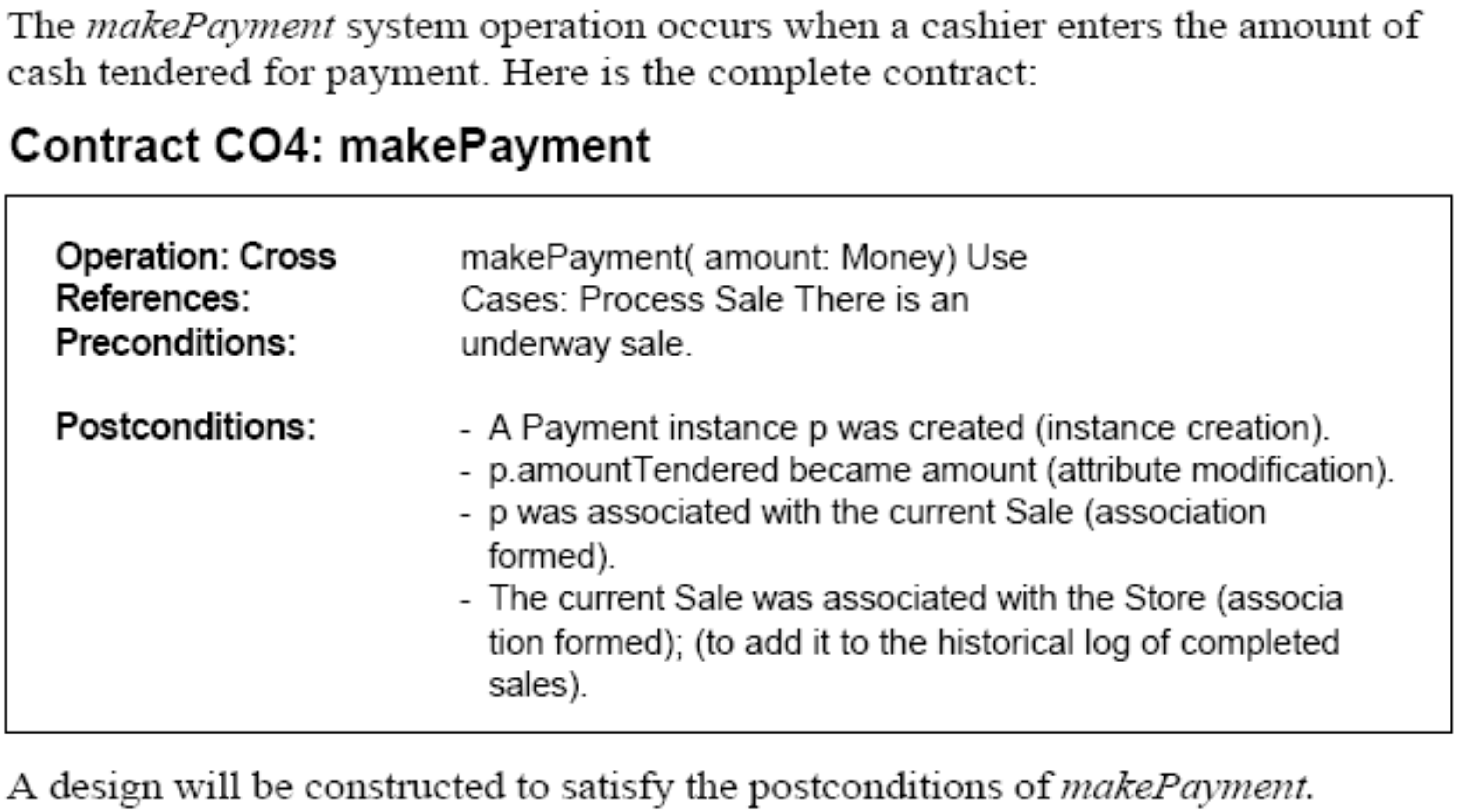
1.1: pr = getPrice

:ProductDescription

Fig. 18.13


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

How to design make Payment



The *makePayment* system operation occurs when a cashier enters the amount of cash tendered for payment. Here is the complete contract:

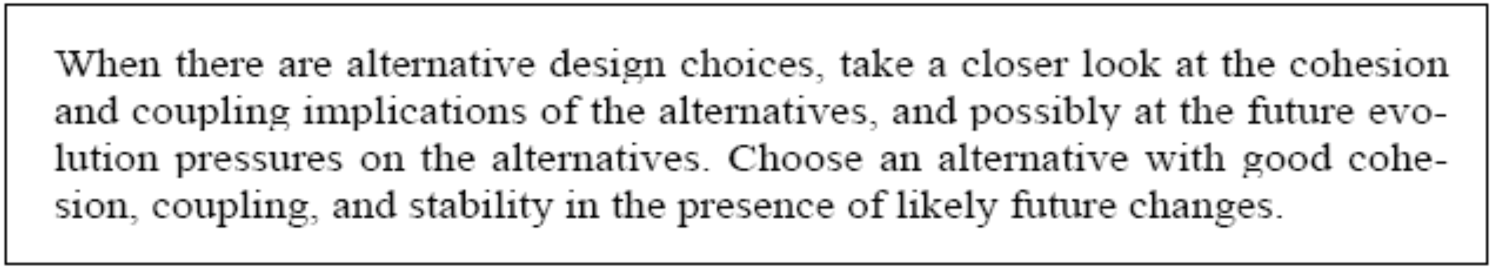
Contract CO4: makePayment

Operation: Cross	makePayment(amount: Money) Use
References:	Cases: Process Sale There is an
Preconditions:	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 (associa tion formed); (to add it to the historical log of completed sales).

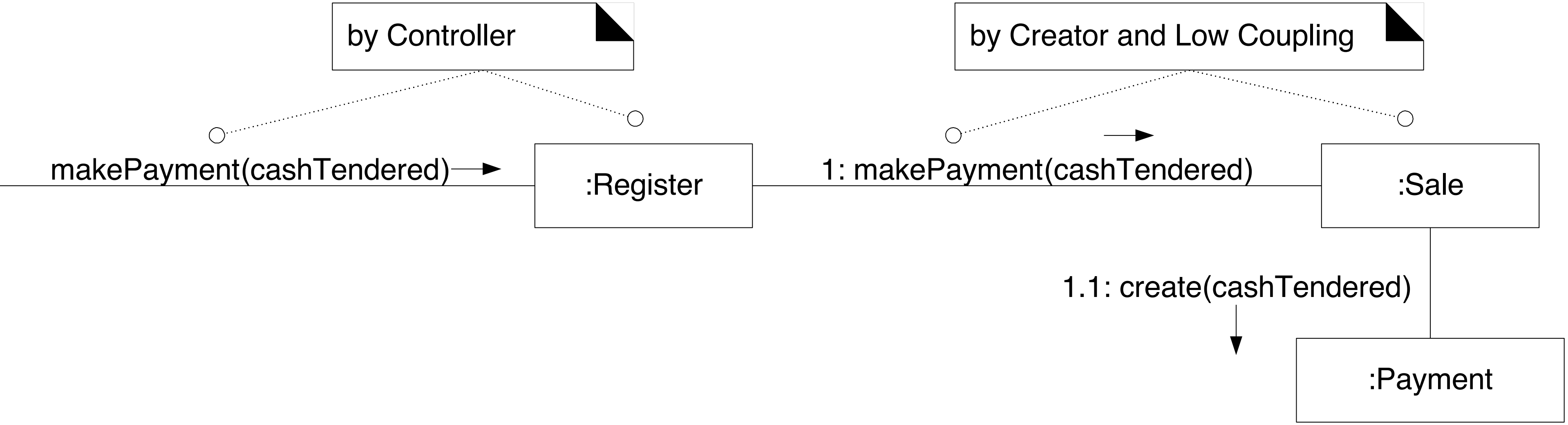
A design will be constructed to satisfy the postconditions of *makePayment*.

How to design makePayment

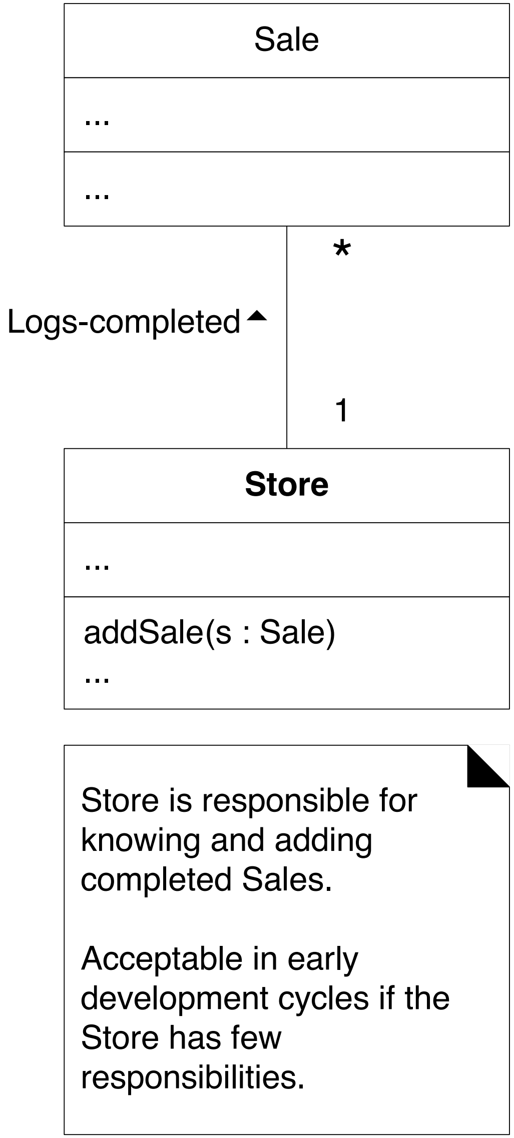
- Creating the payment
  - A payment p is created: consider Creator pattern.
  - Who records, aggregates, most closely uses, or contain Payment?
  - Two candidates: Register, Sale



When there are alternative design choices, take a closer look at the cohesion and coupling implications of the alternatives, and possibly at the future evolution pressures on the alternatives. Choose an alternative with good cohesion, coupling, and stability in the presence of likely future changes.


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

How to design makePayment -- Logging a sale



Sale

...

...

*

Logs-completed ▲

1

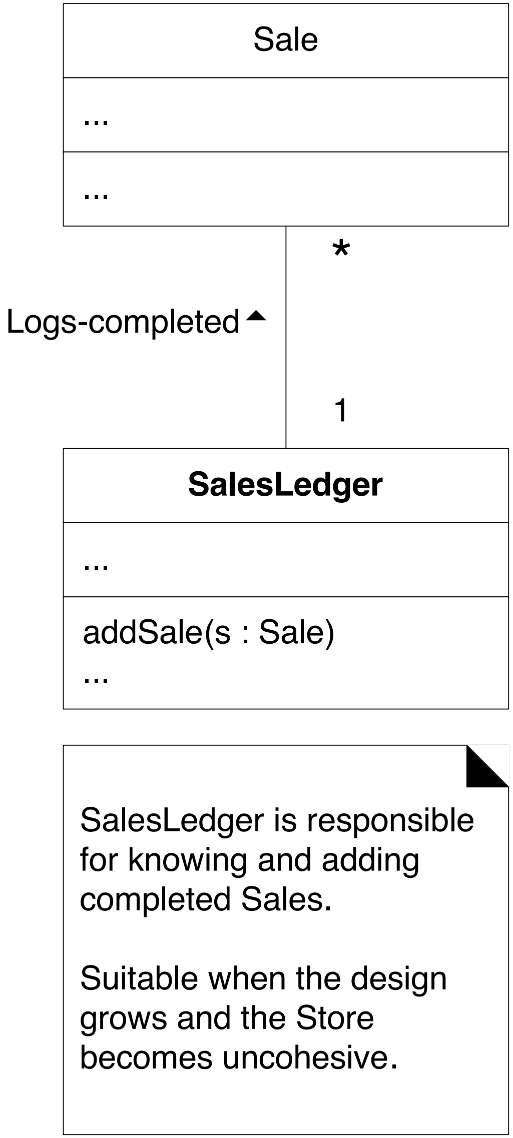
Store

...

addSale(s : Sale)
...

Store is responsible for knowing and adding completed Sales.

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



Sale

...

...

*

Logs-completed ▲

1

SalesLedger

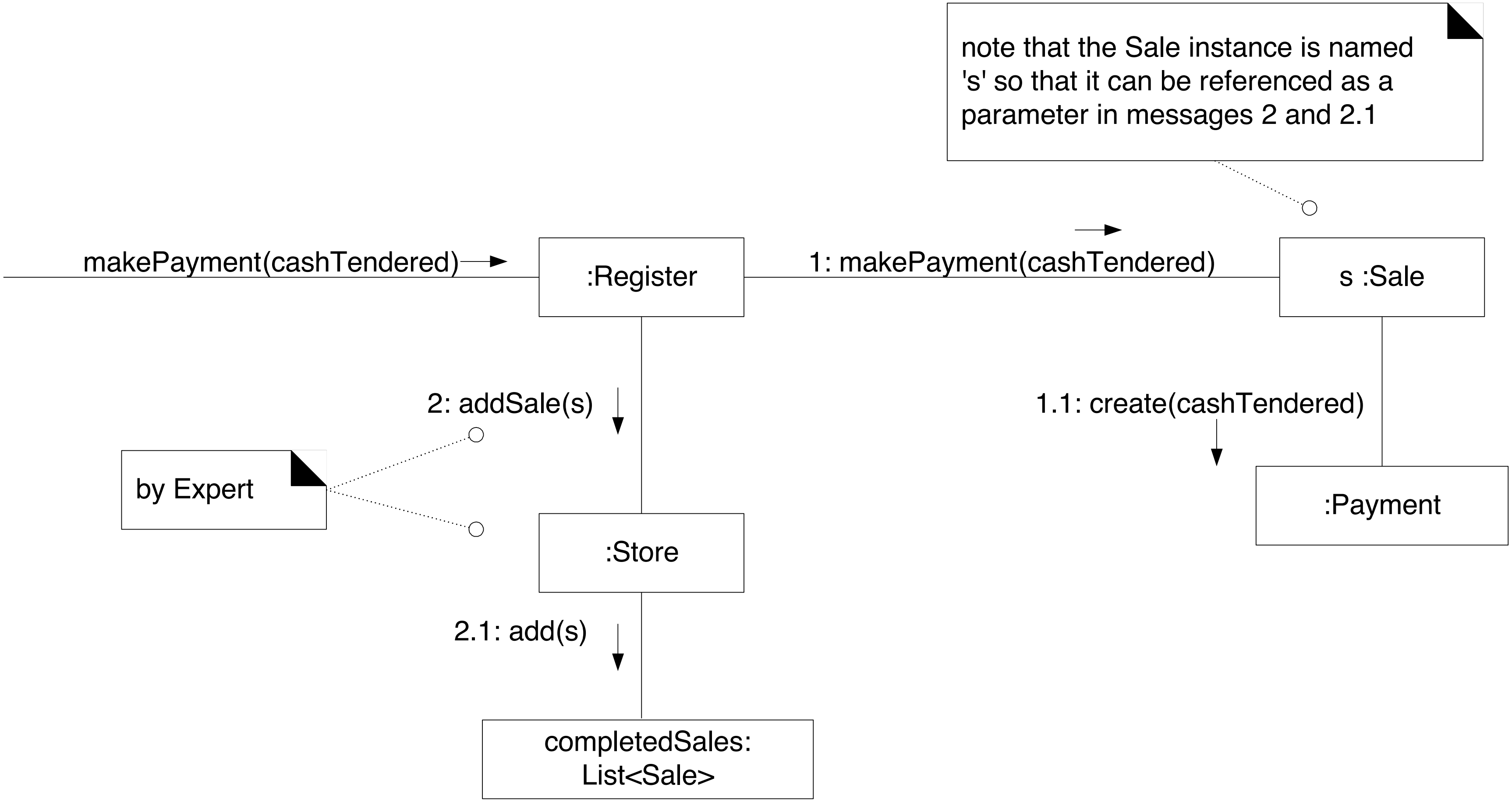
...

addSale(s : Sale)
...

SalesLedger is responsible for knowing and adding completed Sales.

Suitable when the design grows and the Store becomes uncohesive.

Fig. 18.15 Logging a completed sale


note that the Sale instance is named 's' so that it can be referenced as a parameter in messages 2 and 2.1

by Expert

makePayment cont.   Calculating Balance

- Process use case the balance be printed on a receipt
- MVS : we should not be concerned with how it will be printed, but we must ensure that it is known
- Who is responsible for knowing the balance?

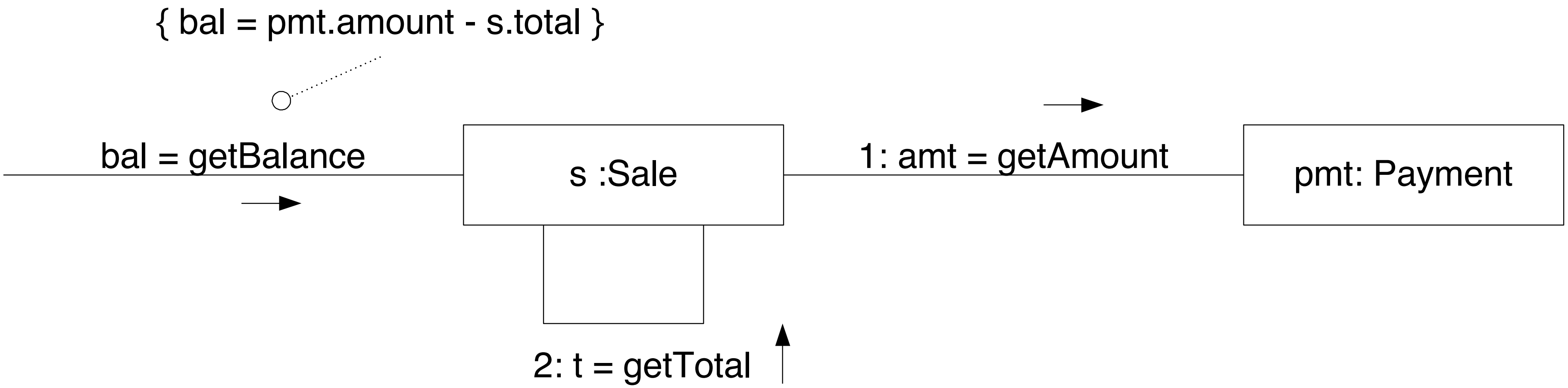
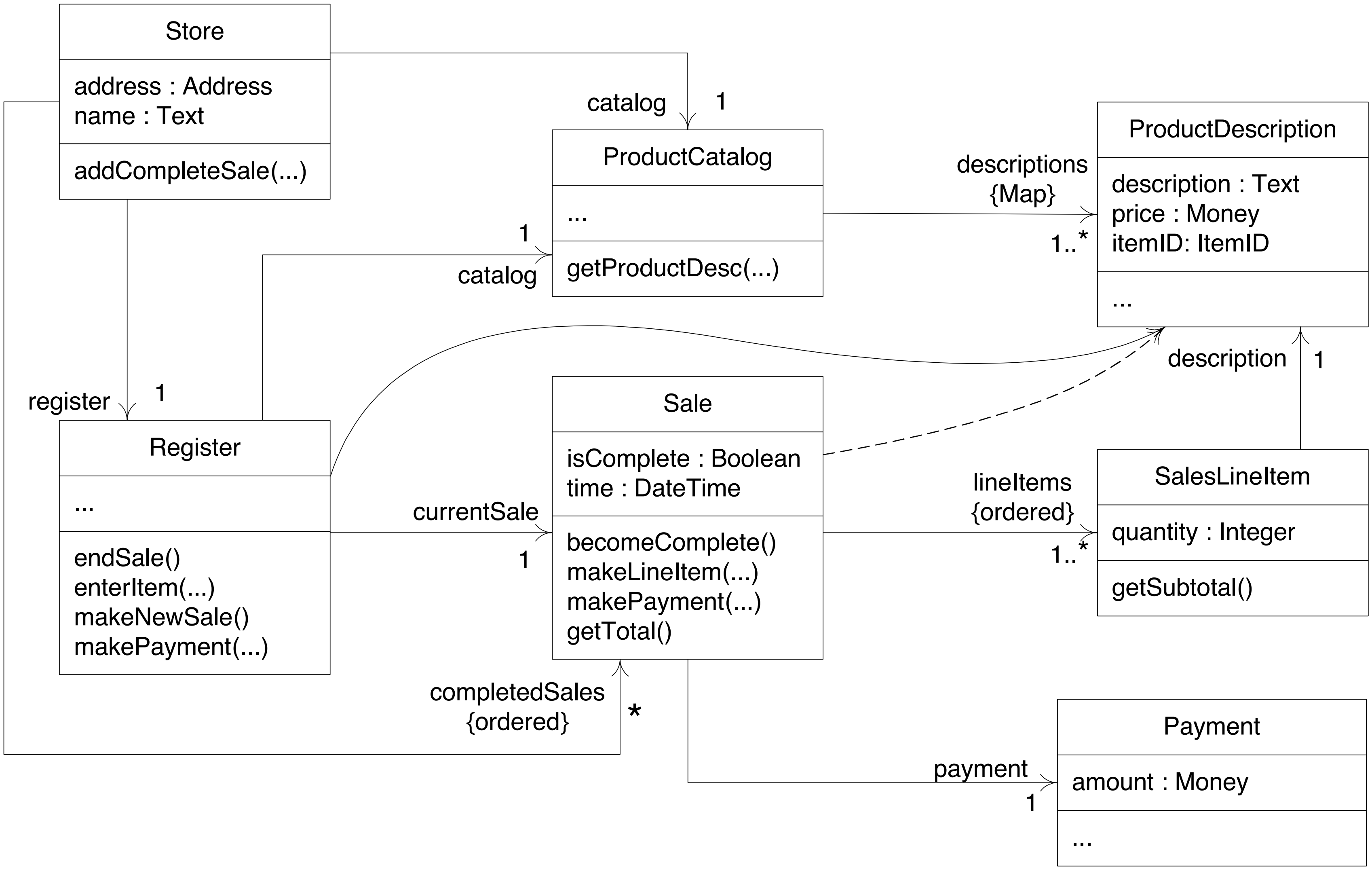
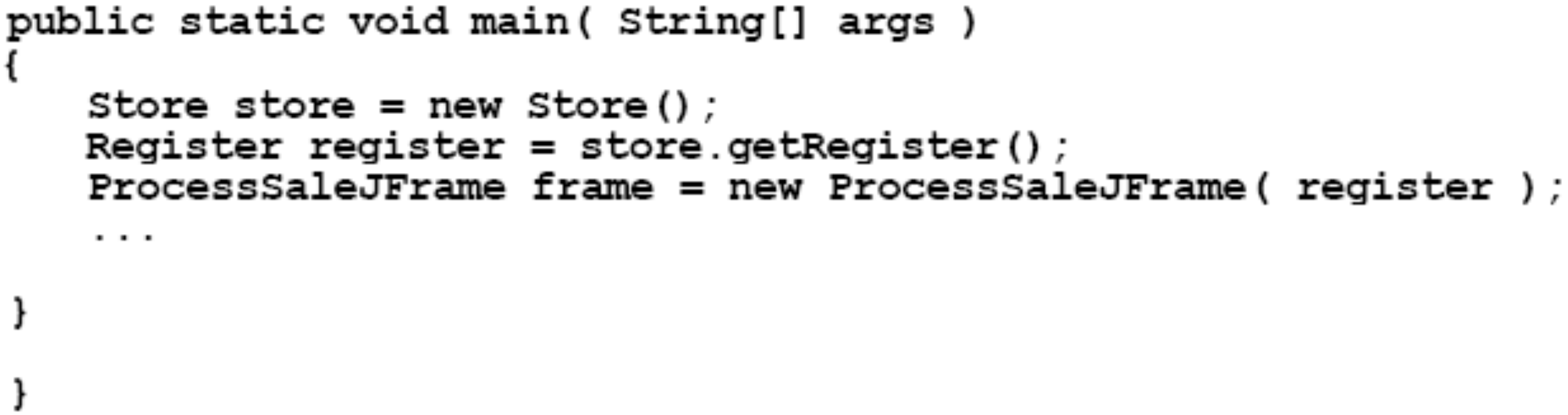


Fig. 18.17 A more complete DCD



Connecting UI Layer to Domain Layer

- Common designs for objects in UI Layer to obtain visibility to objects in Domain Layer:
  - An initialized object (for example a Factory) called from starting method (e.g., Java main) creates both a UI and a Domain object and passes the domain object to the UI.
  - A UI object retrieves the domain object from a well-known source, such as a factory object that is responsible for creating domain objects.



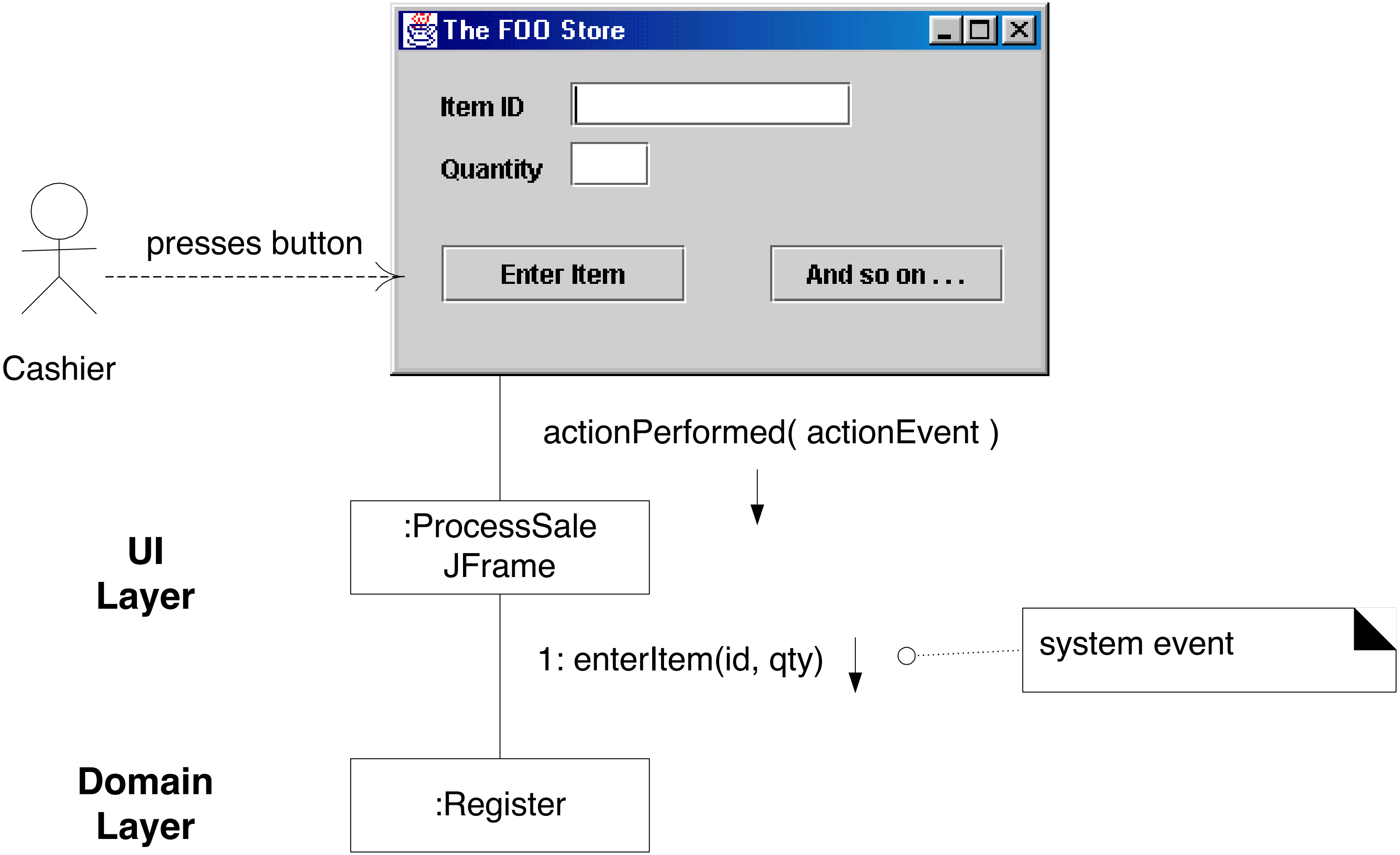
```
public static void main( String[] args )
{
    Store store = new Store();
    Register register = store.getRegister();
    ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
    ...

}

}
```

- During start up or initialize system operation, **an initial domain object** , or a set of peer initial domain objects are first created.
  - This creation can be done in main, or in a Factory method
- Initial domain object is responsible for creation of its child domain objects
- Domain controller object reference is passed to UI objects

Fig. 18.18 Connecting UI to Domain layers



Cashier

presses button

actionPerformed(actionEvent)

UI Layer

:ProcessSale JFrame

1: enterItem(id, qty)

system event

Domain Layer

:Register

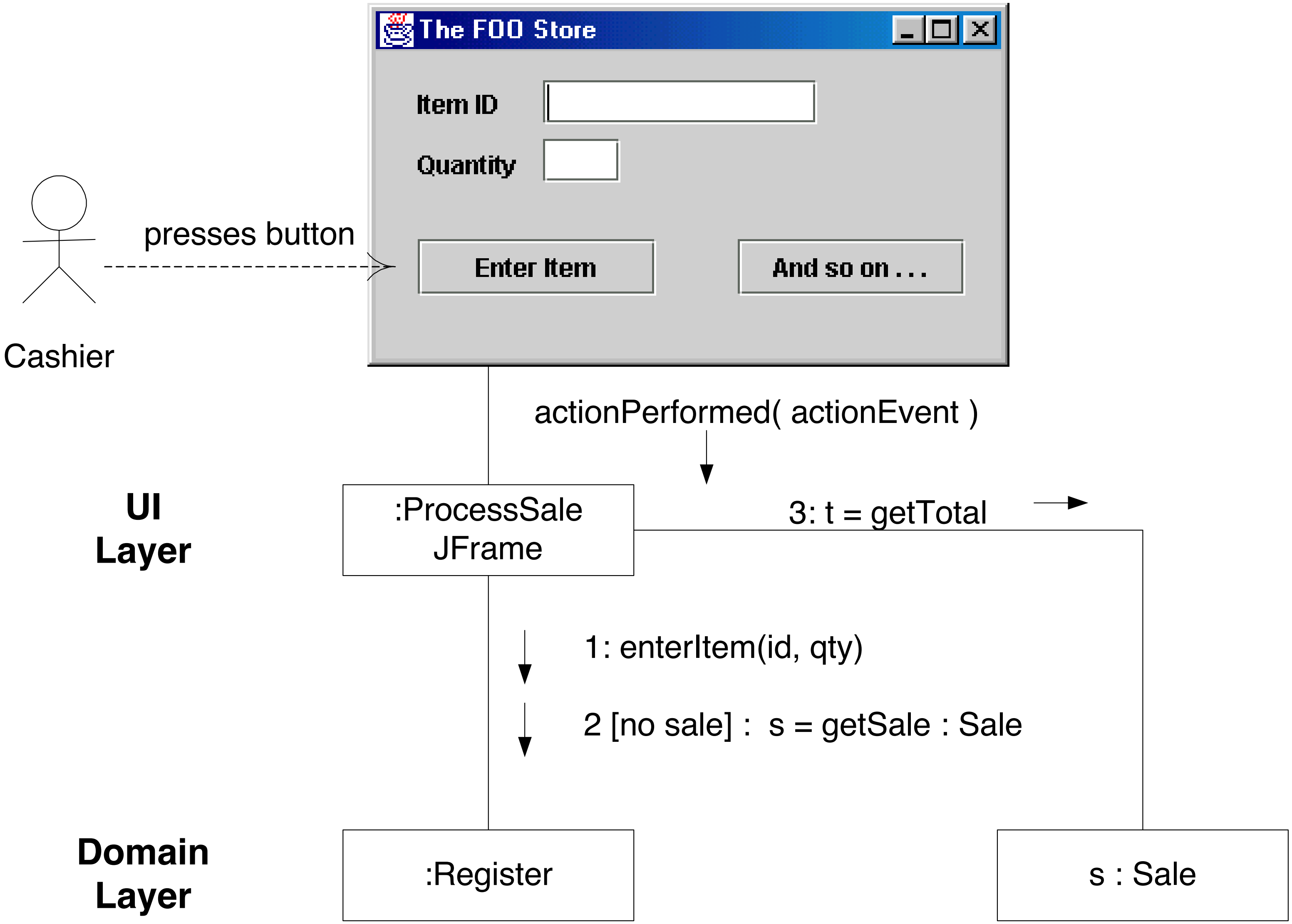
Displaying output

- Once the UI object has a connection to the Register instance (the façade controller), it can forward system event messages, such as enterItem
- For enterItem operation: we want the window to show the running total after each entry.

Design Solutions for connecting UI to Domain Layer

- Add a getTotal method to Register.
  - Delegates to Sale
  - Possible advantage: lower coupling from UI to domain layer (UI knows only Register)
  - But it expands Register interface and makes it less cohesive
- A UI asks for a reference to the current Sale object, and then it directly sends messages to Sale
  - Increased coupling
  - However, coupling to unstable things is a real problem
  - For this case, Sale object can be made an integral part of the design – which is reasonable.

Fig. 18.19



presses button

Cashier

actionPerformed(actionEvent)

UI Layer

:ProcessSale JFrame

3: t = getTotal

1: enterItem(id, qty)

2 [no sale] : s = getSale : Sale

Domain Layer

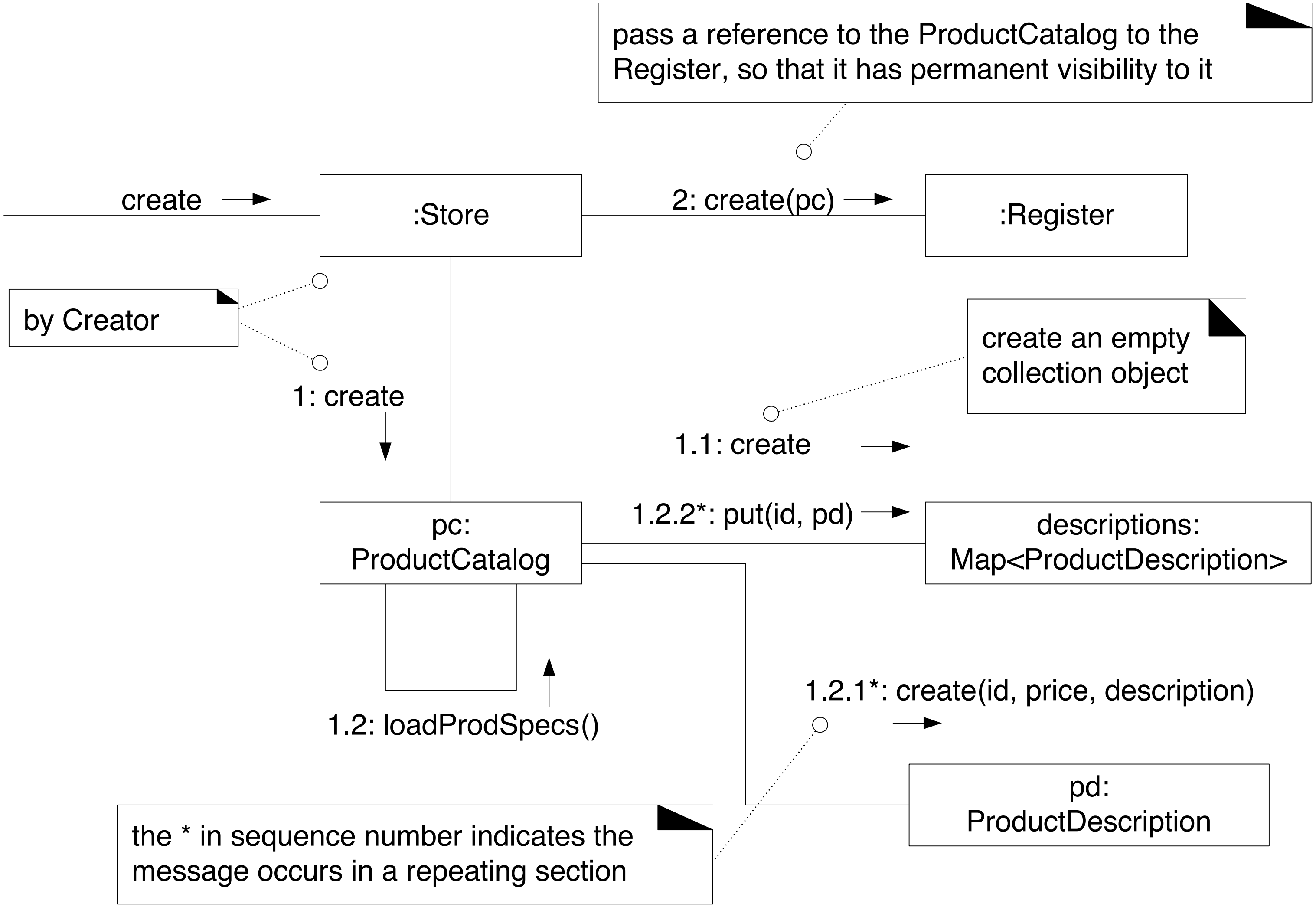
:Register

s : Sale

Choosing Initial Domain Object

- Choose as an initial domain object a class at or near the root of the containment or aggregation hierarchy of domain objects. This may be a facade controller, such as *Register,* or some other object considered to contain all or most other objects, such as a *Store.*

Fig. 18.20 Store.create design



pass a reference to the ProductCatalog to the Register, so that it has permanent visibility to it

by Creator

create an empty collection object

the * in sequence number indicates the message occurs in a repeating section

Use Case Realizations for the Monopoly Iteration

Fig. 18.21 Domain Model for Monopoly

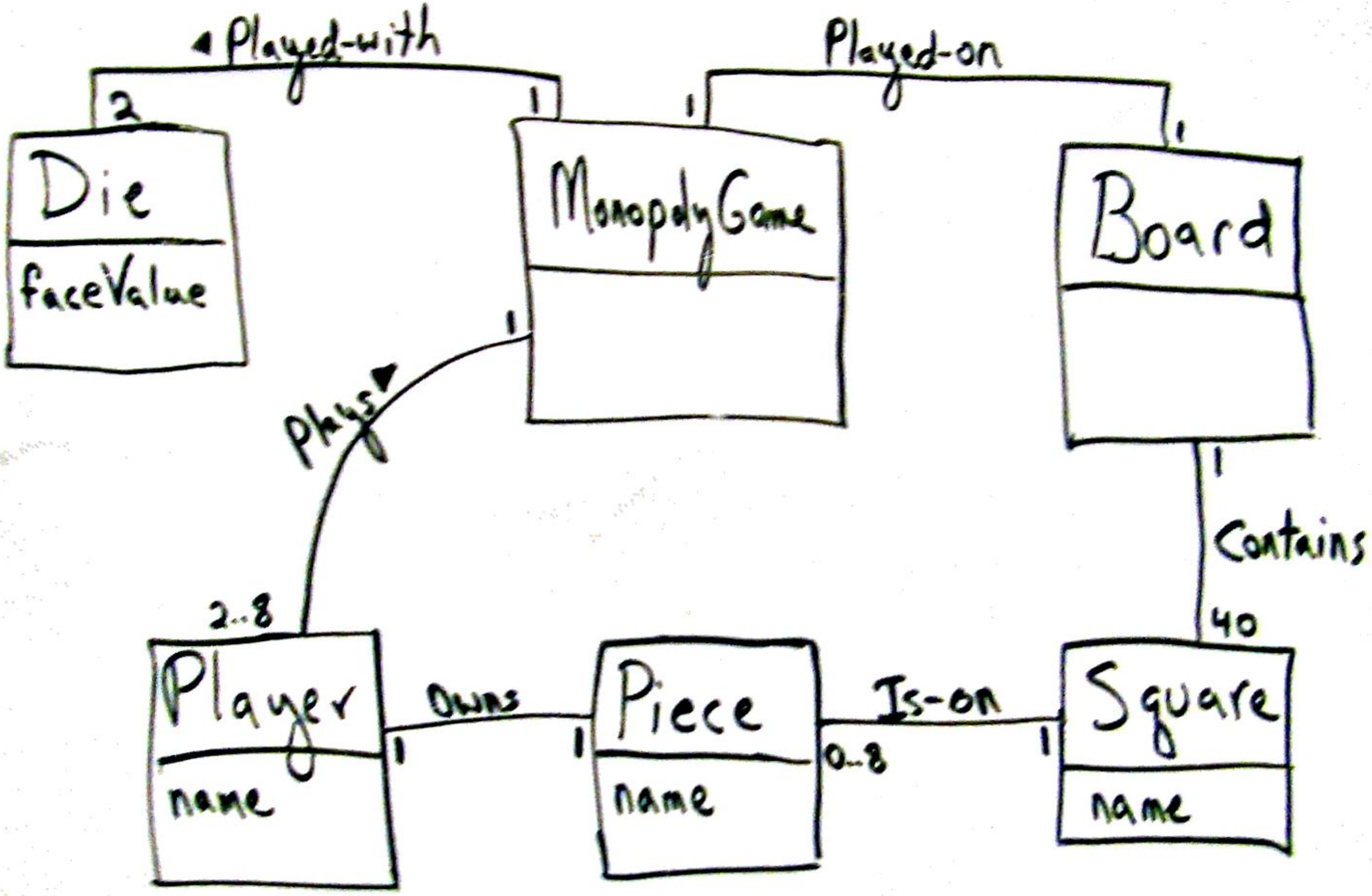
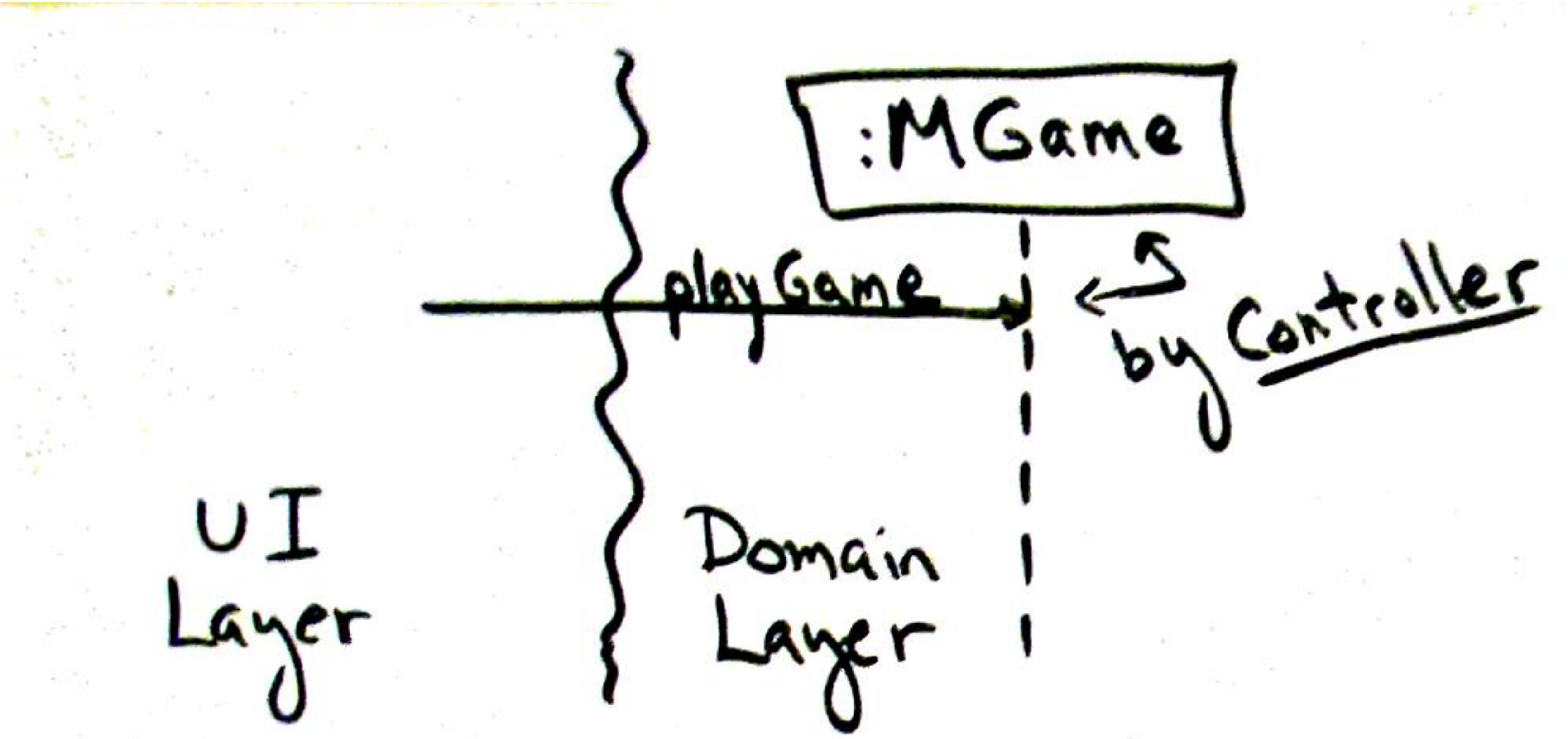


Fig. 18.22 Applying controller to the playGame system operation

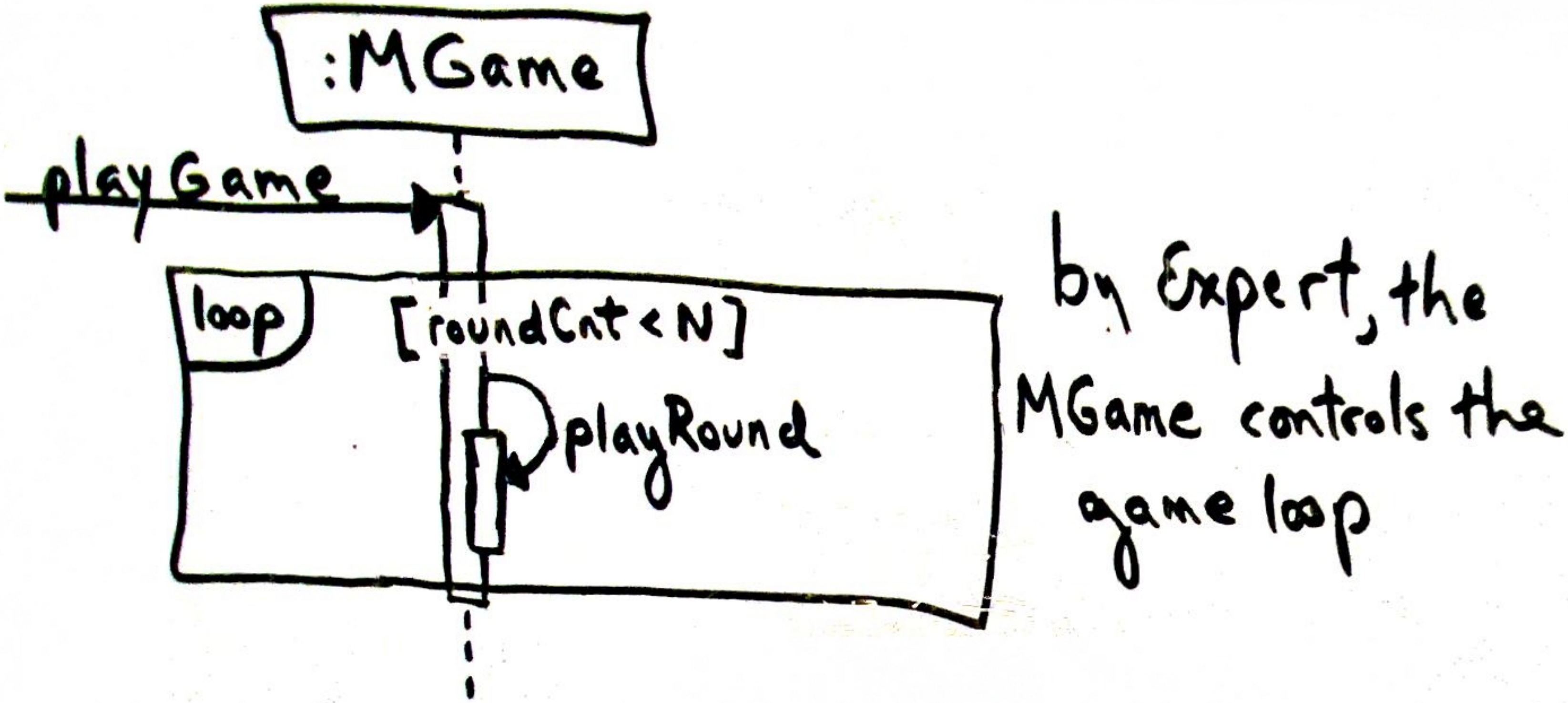


The Game-Loop Algorithm

```
for N rounds
    for each Player  p
        p takes a turn
```

- Who is responsible for controlling game loop
- Doing responsibility : Expert
  - What information is needed for the responsibility?
  - MonopolyGame is a good candidate

Fig. 18.23 Game Loop



by Expert, the MGame controls the game loop

Good OO method design encourage small methods with a single purpose

Taking a Turn

- Who takes turn?
- Taking a turn means
  - Calculating a rnd # between 2-12
    - LRG: we’ll create a Die object
    - Expert: Die should able to roll itself
  - Calculating new square
    - LRG: Board knows its squares
    - Expert: Board will be responsible
  - Moving player’s piece
    - LRG: Player know its piece
    - Piece knows its square
    - Expert: Piece will set its new location (but it will receive from its owner Player)
- Who coordinates
- Problem of Visibility

Fig. 18.24 Player takes a turn by Expert

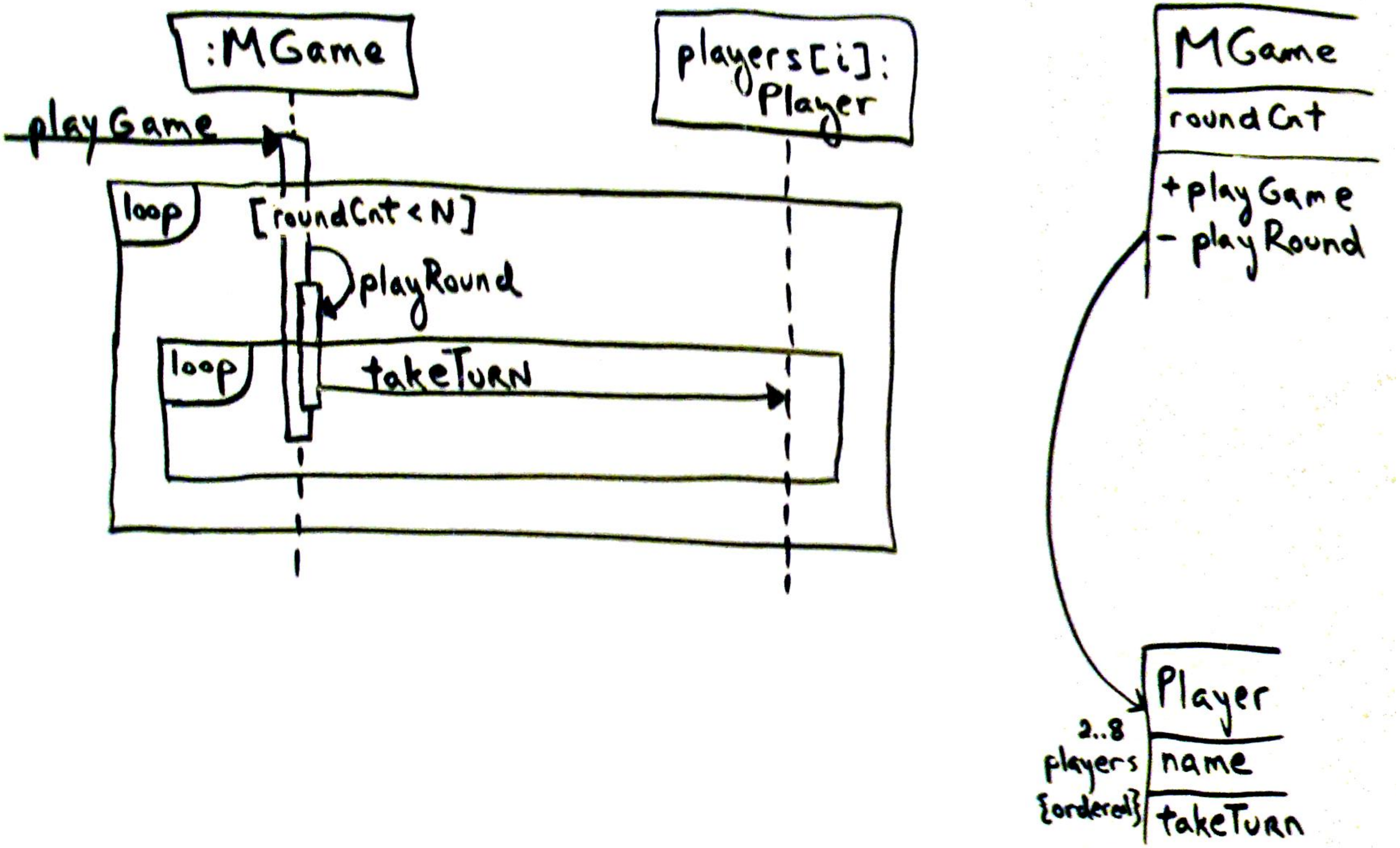


Fig. 18.25 Dynamic Design for playGame

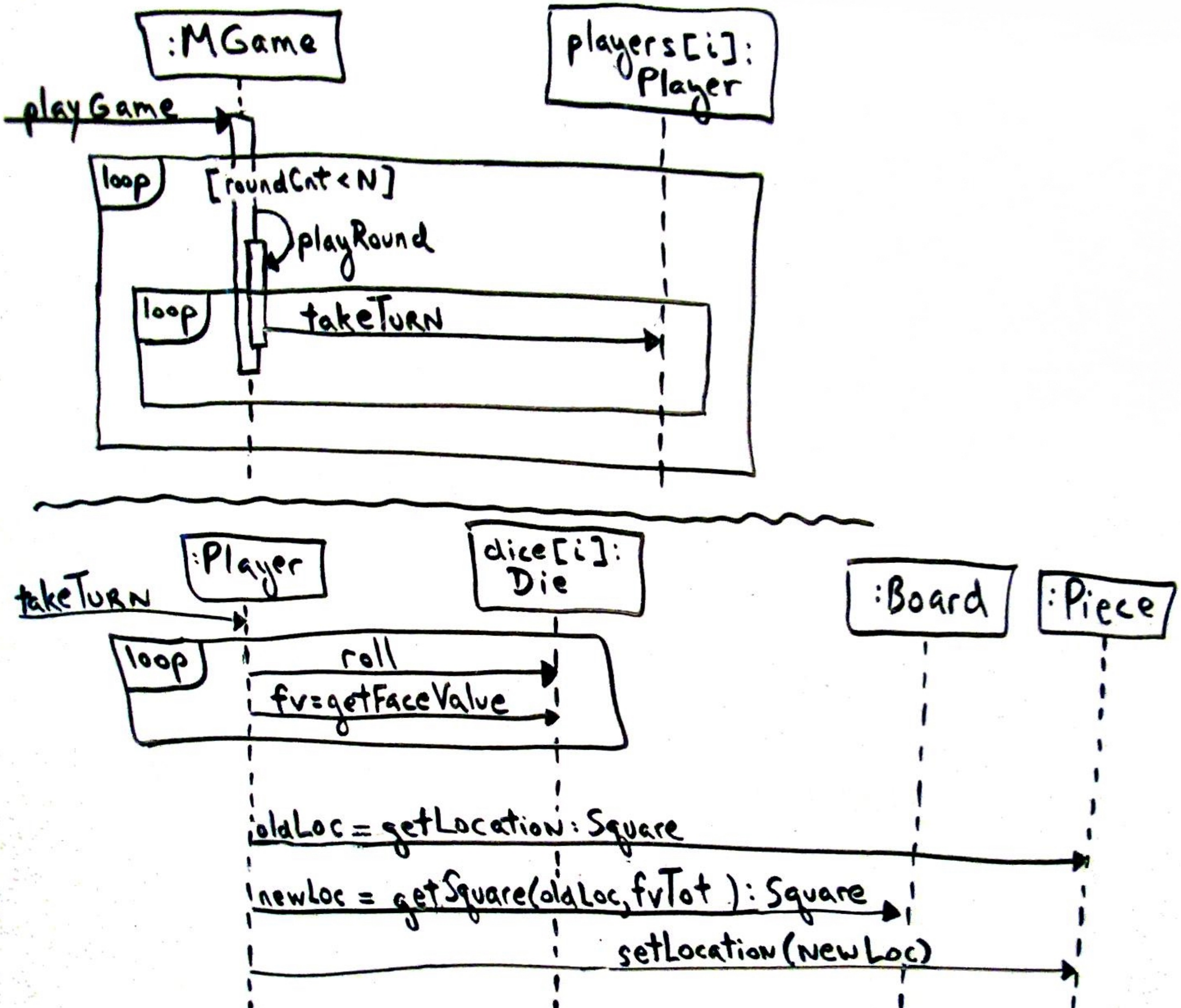
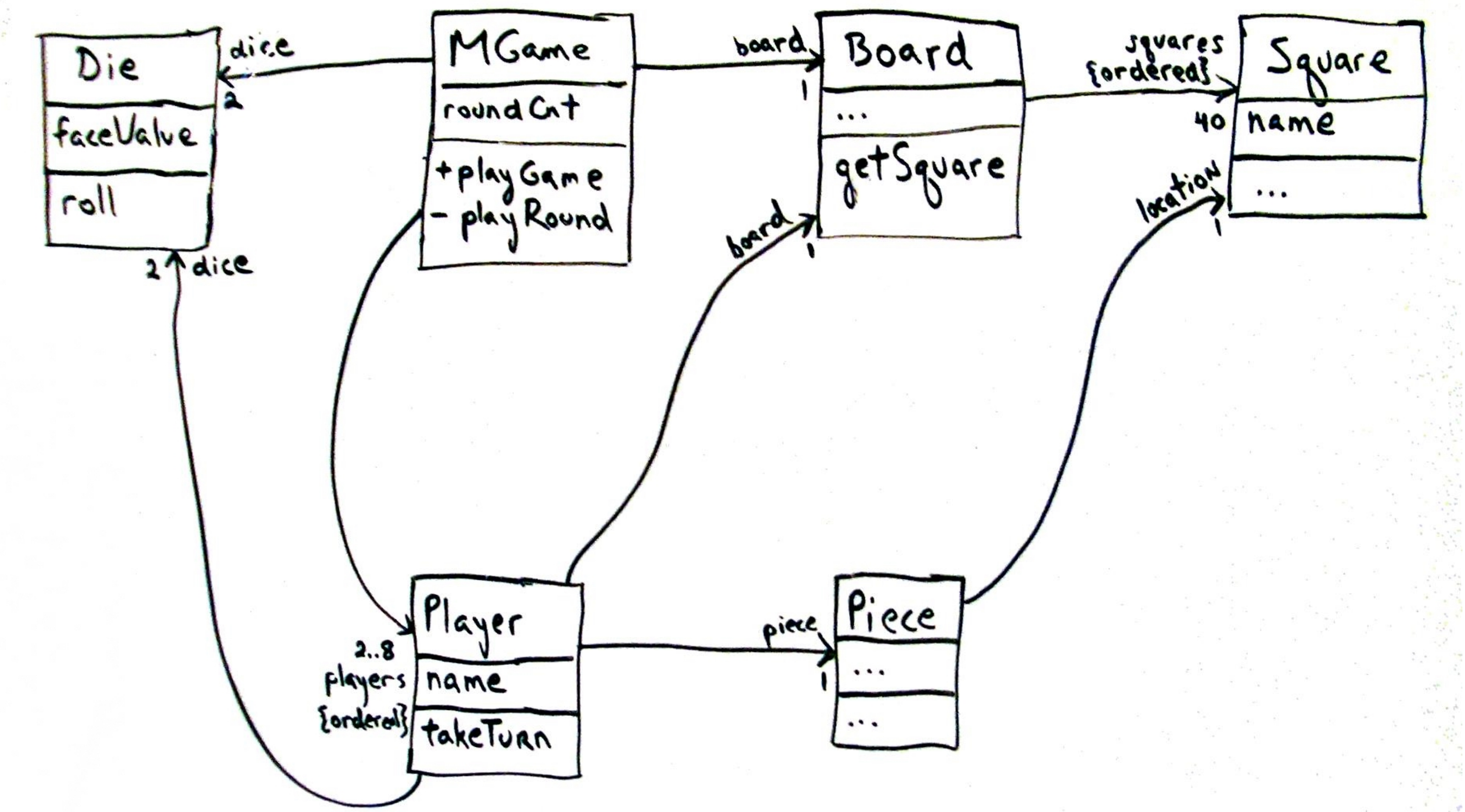


Fig. 18.26 Static Design for playGame



Command-Query Separation Principle

Style #1

```
public void roll() {
   faceValue = random….
}
public int getFaceValue() {
Return faceValue;
}
```

Style #2

```
public int roll() {
   faceValue = random….
   return faceValue;
}
```

- A command method that performs an action (updating, coordinating) often has side effects such as changing the state of objects and is void (no return value); or
- A query that returns data to the caller and has no side effects – it should not permanently change the state of any objects.

A method should not be both

Fig. 18.27 Creation Dependencies

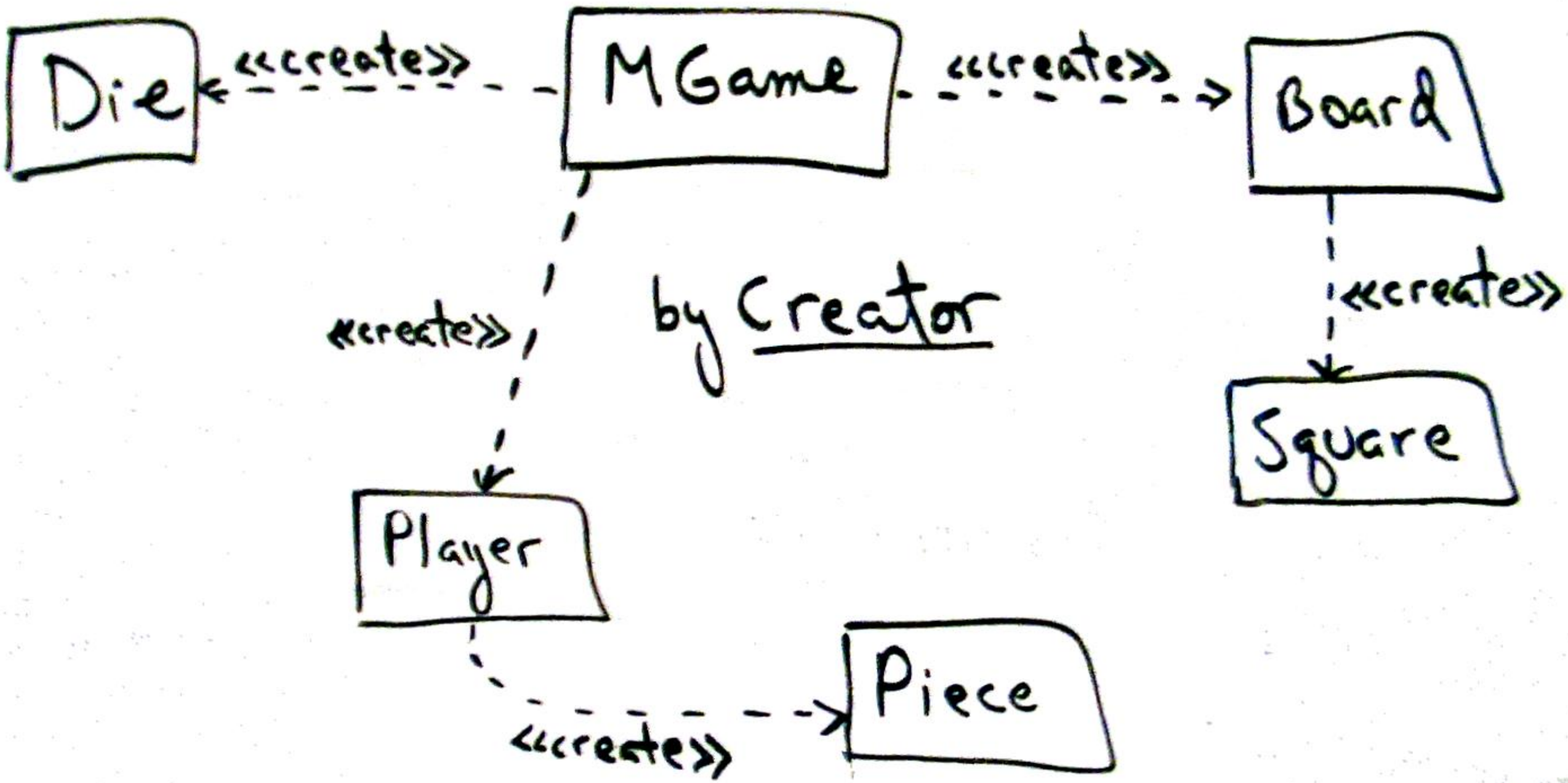
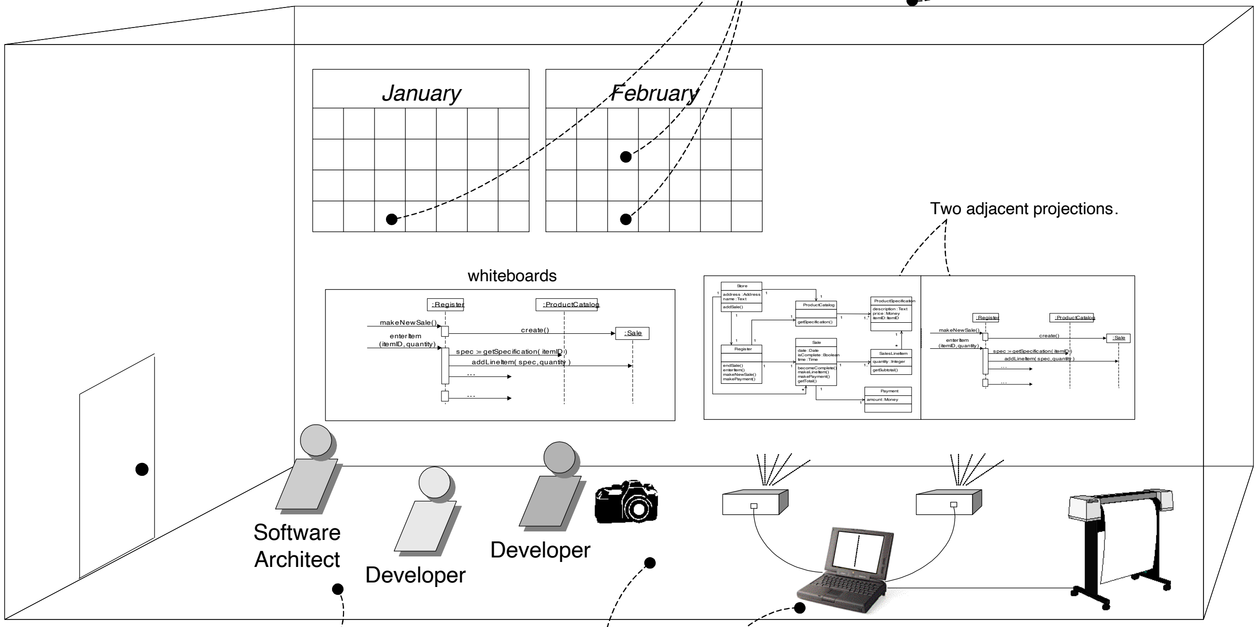


Fig. 18.28

When
Near the beginning of each iteration, for a "short" period before programming.

Where
In a project room with lots of support for drawing and viewing drawings.



Who
Perhaps developers will do some design work in pairs. The software architect will collaborate, mentor, and visit with different design groups.

How: Tools
Software: A UML CASE tool that can also reverse engineer diagrams from code.

Hardware:
- Use two projectors attached to dual video cards.
- For whiteboard drawings, perhaps a digital camera.
- To print noteworthy diagrams for the entire team, a plotter for large-scale drawings to hang on walls.