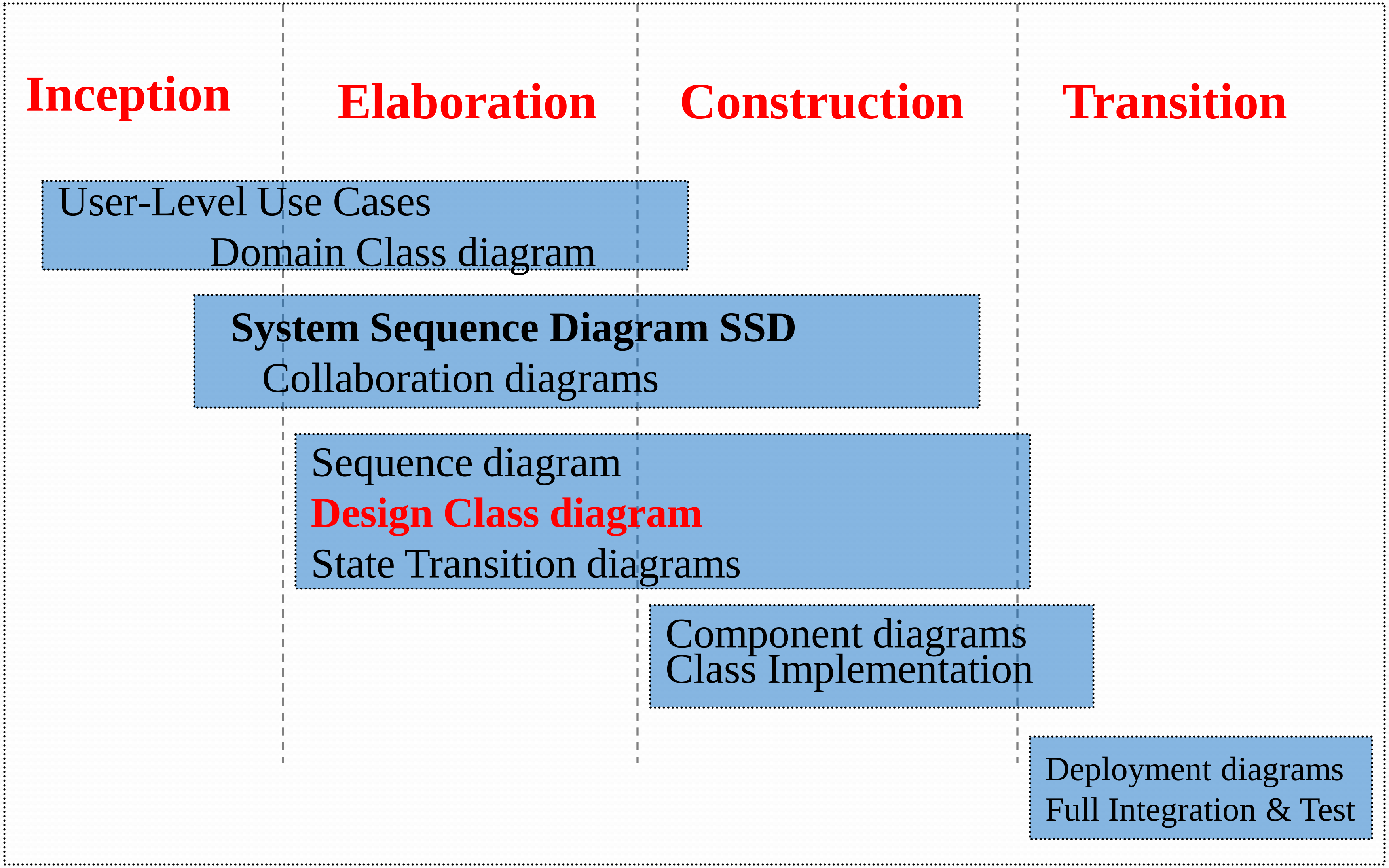
COMP 3711

OOD

Static Object Modeling
Design Class Diagram

Larman Chapter 16

UML And UP



Inception

Elaboration

Construction

Transition

User-Level Use Cases

Domain Class diagram

System Sequence Diagram SSD

Collaboration diagrams

Sequence diagram

Design Class diagram

State Transition diagrams

Component diagrams

Class Implementation

Deployment diagrams

Full Integration & Test

Object Design

"After identifying your requirements, documenting in Use Cases, creating a Domain Model, SSD and Sequence Diagram .....

What is next?

"The next task is to add methods to the software classes, and define the messaging between the objects to fulfill the requirements"

Responsibility Driven Design - RDD

- Think of software objects as having responsibilities → what they do

- Responsibilities are related to the obligations or behaviour of an object in terms of its role

- Responsibilities are implemented by means of Methods acting alone or collaborating

- RDD – a general *Metaphore* of a community of collaborating responsible objects

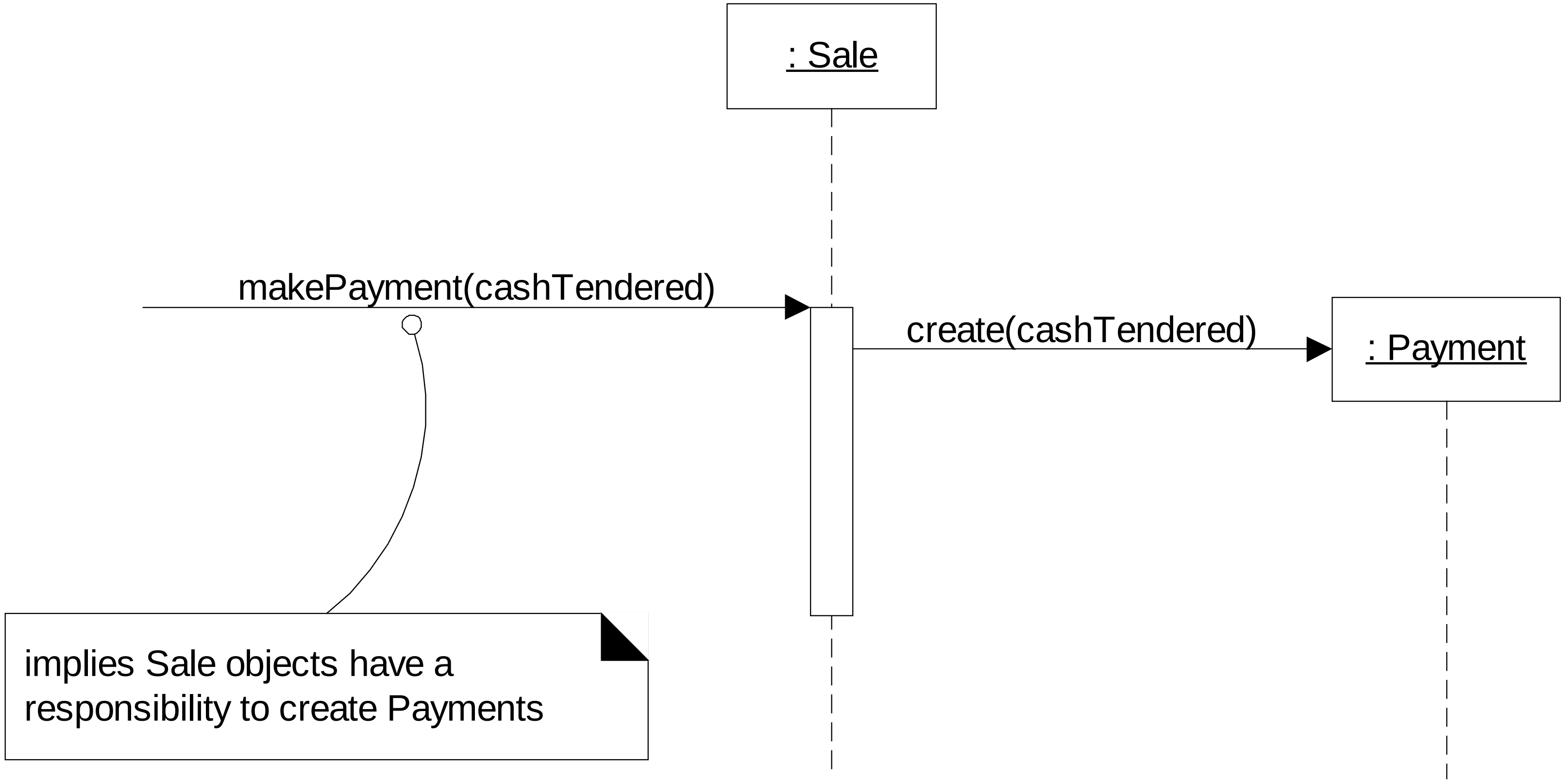
Two Types Of Responsibilities

- Doing Responsibilities
  - Creating an object or doing a calculation
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects
- Knowing Responsibilities
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things that can be derived or calculated

Responsibilities - Interaction Diagrams

- Show objects and the messages in-between
- UML Interaction Diagrams include:
  - Sequence diagrams
    - Have time on the Y axis
  - Collaboration diagrams
    - Focus is more on way the objects interact
- Record assignment of responsibilities

Sequence Diagram Example



implies Sale objects have a responsibility to create Payments

Design - Think Object

- Assigning responsibilities
- Granularity of responsibility influences how it is assigned
- What methods belong where?
- How objects should interact?

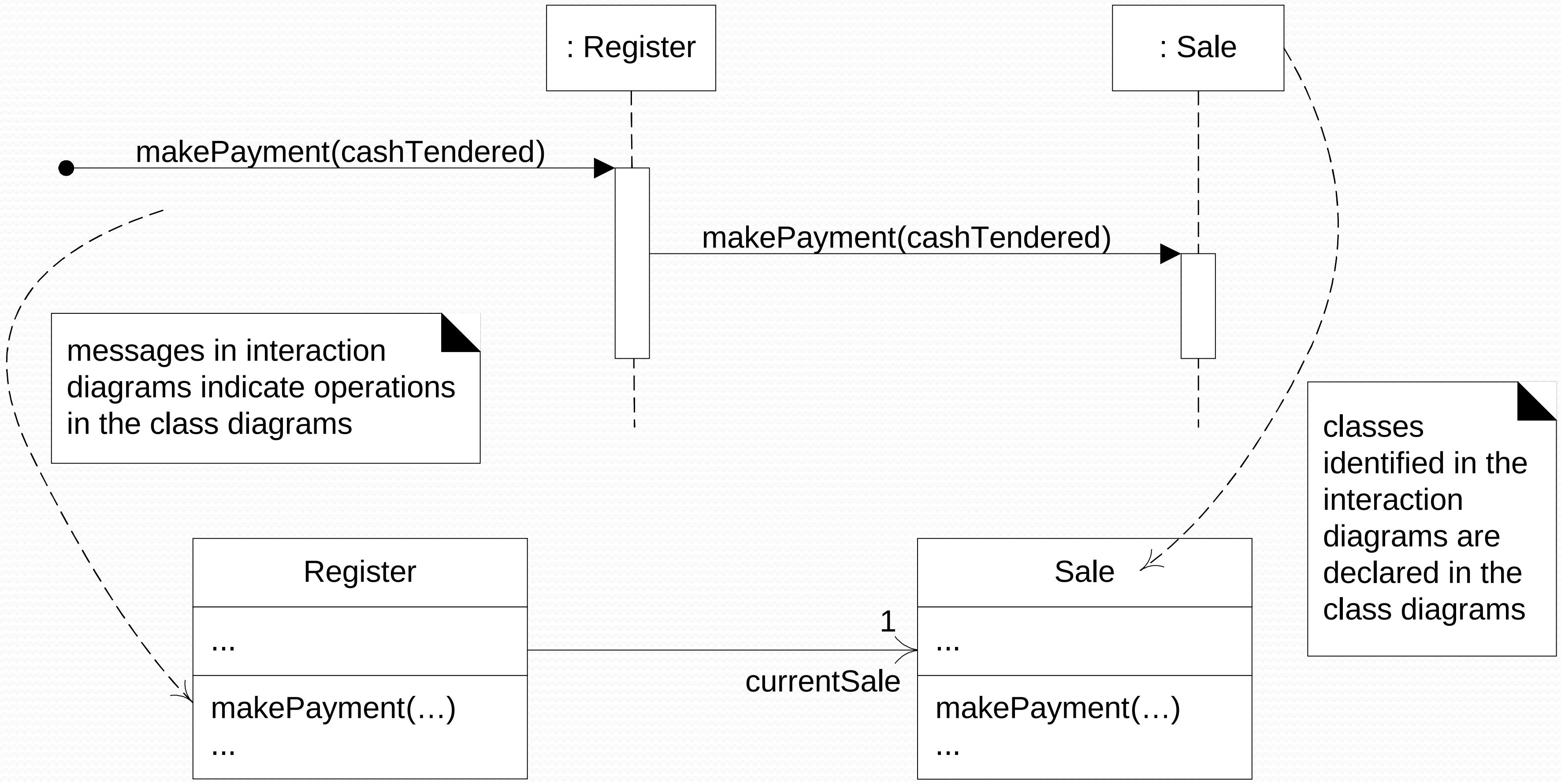
UML Object Modeling

- Domain Model
  - Use Case / Use Case Diagrams
  - Conceptual Classes Diagrams
  - SSD
- Design Model
  - Design Classes Diagrams (DCD)
  - Interaction Diagrams
  - Package Diagrams

Steps In Creating DCDs

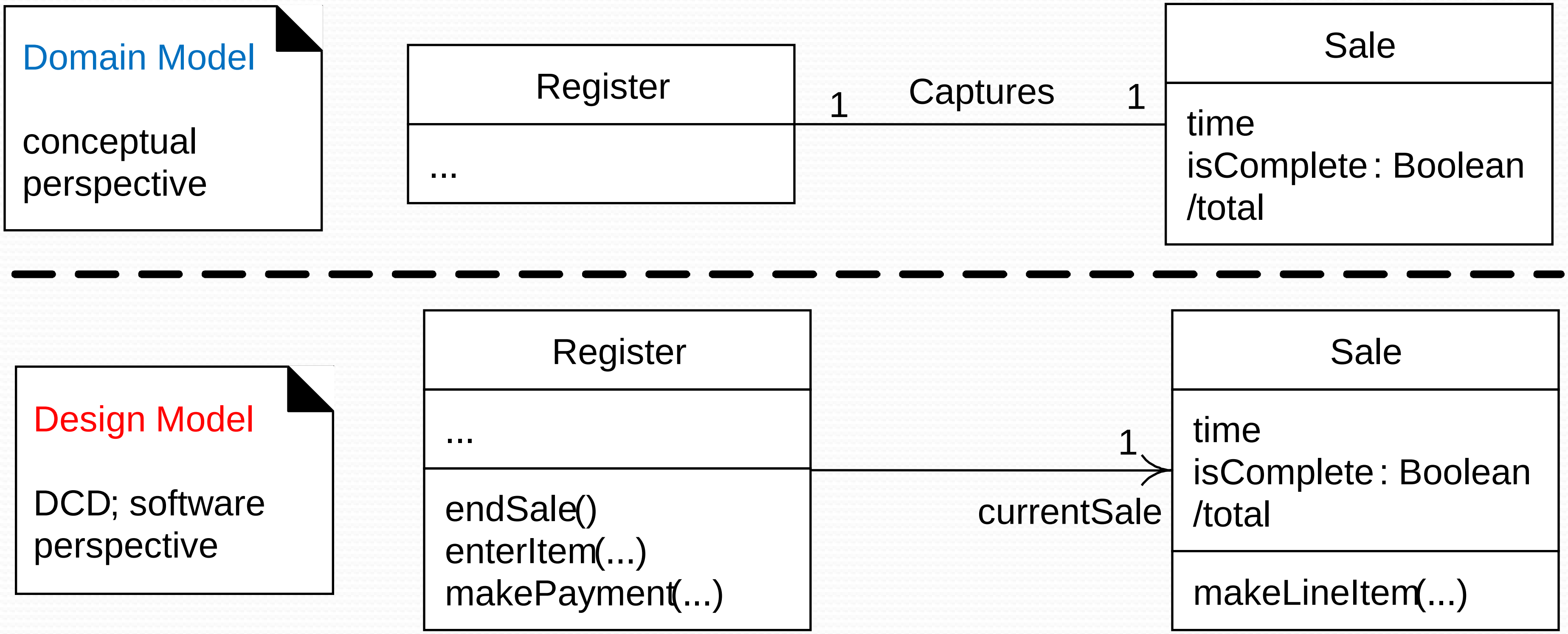
- Start with the conceptual class diagram
- Determine which classes need to be broken down into software classes
- Identify attributes, many of which carry over from the conceptual class diagram
- Use the interaction diagrams to identify methods for each class

Example From Sequence Diagram


: Register
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
messages in interaction diagrams indicate operations in the class diagrams
classes identified in the interaction diagrams are declared in the class diagrams
Register
...
makePayment(…)
...
1
currentSale
Sale
...
makePayment(…)
...

Larman Fig 16.21

Conceptual vs Design Class Diagrams



Domain Model

conceptual perspective

Register

...

1 Captures 1

Sale

time
isComplete : Boolean
/total

Design Model

DCD; software perspective

Register

...

endSale()
enterItem(...)
makePayment(...)

1

currentSale

Sale

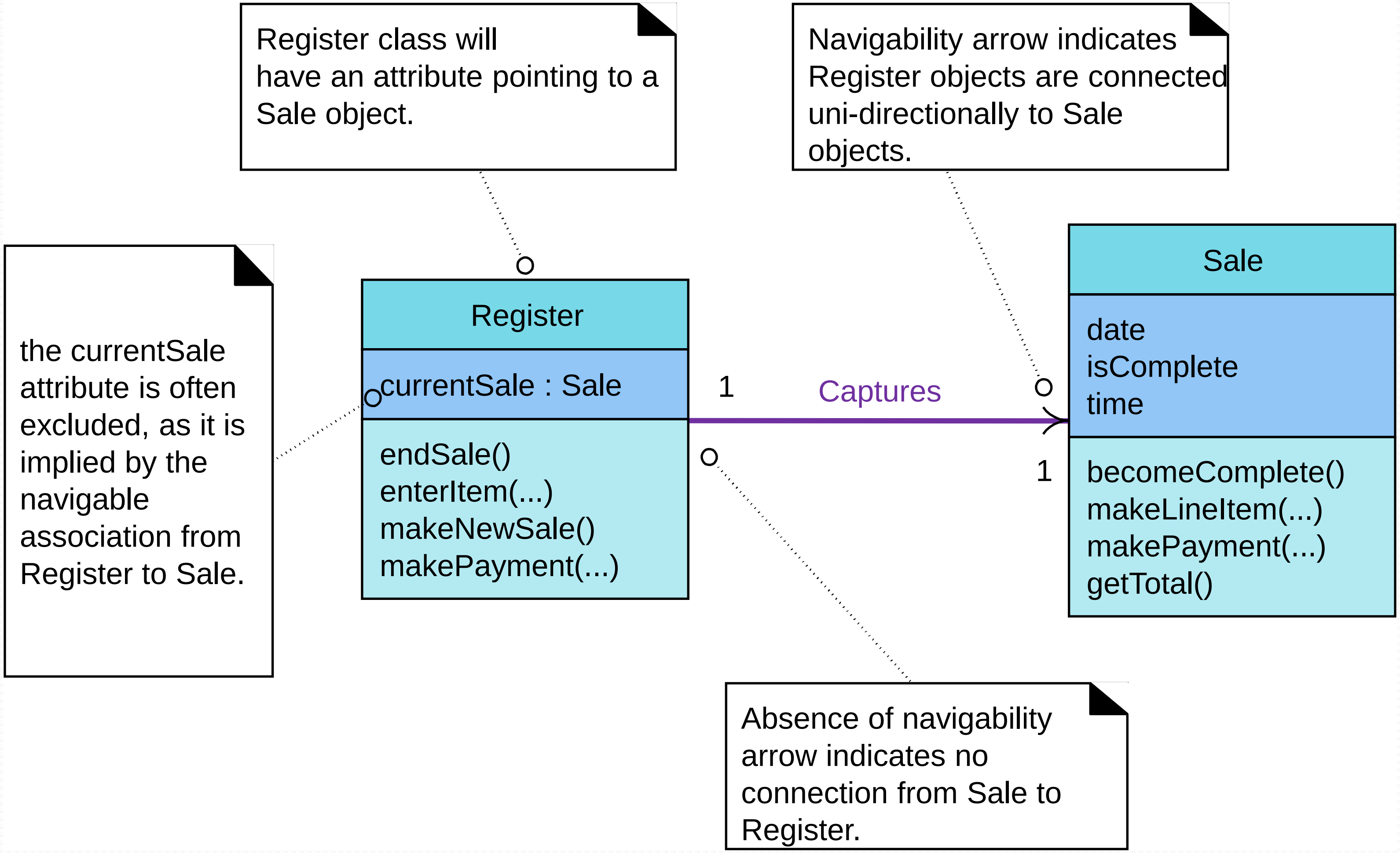
time
isComplete : Boolean
/total

makeLineItem(...)

Design Class Diagrams - DCD

- The classes in the DCD are software classes include attributes and methods and may include interfaces
- DCD does <u>not</u> have to include boundary and control objects
- DCD can show all classes for a simple system
- In industry, a DCD shows all classes in each subsystem or major component, and the other subsystems are represented by interfaces

DCD - Example



Register class will have an attribute pointing to a Sale object.

Navigability arrow indicates Register objects are connected uni-directionally to Sale objects.

the currentSale attribute is often excluded, as it is implied by the navigable association from Register to Sale.

Register

currentSale : Sale

endSale()
enterItem(...)
makeNewSale()
makePayment(...)

1 Captures 1

Sale

date
isComplete
time

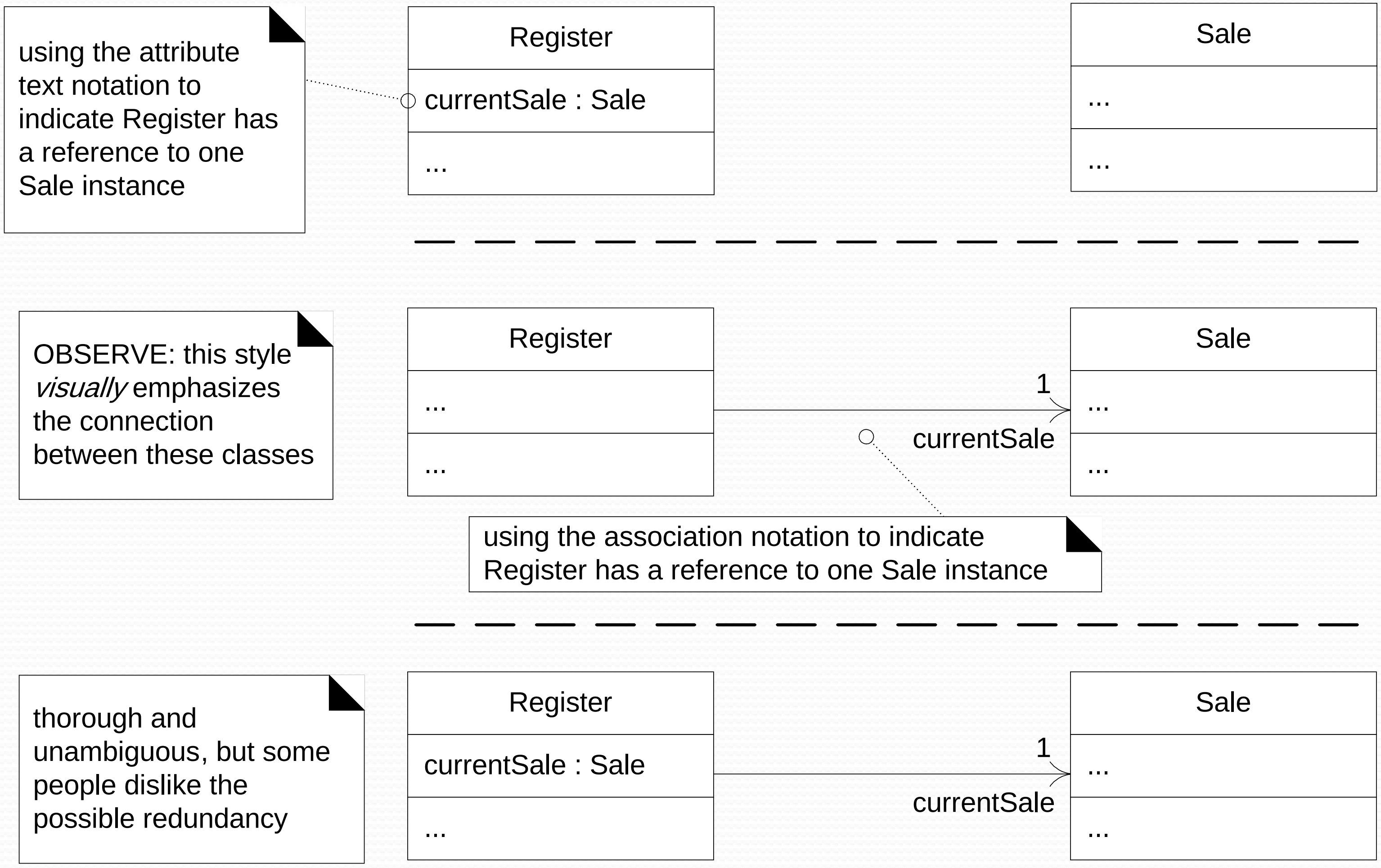
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()

Absence of navigability arrow indicates no connection from Sale to Register.

Typical Information On DCD

- Classes, associations, and attributes
- Interfaces, with their operations and constants indicated
- Methods
- Attribute types
- Navigability
- Dependencies

DCD - Attributes And Association



using the attribute text notation to indicate Register has a reference to one Sale instance

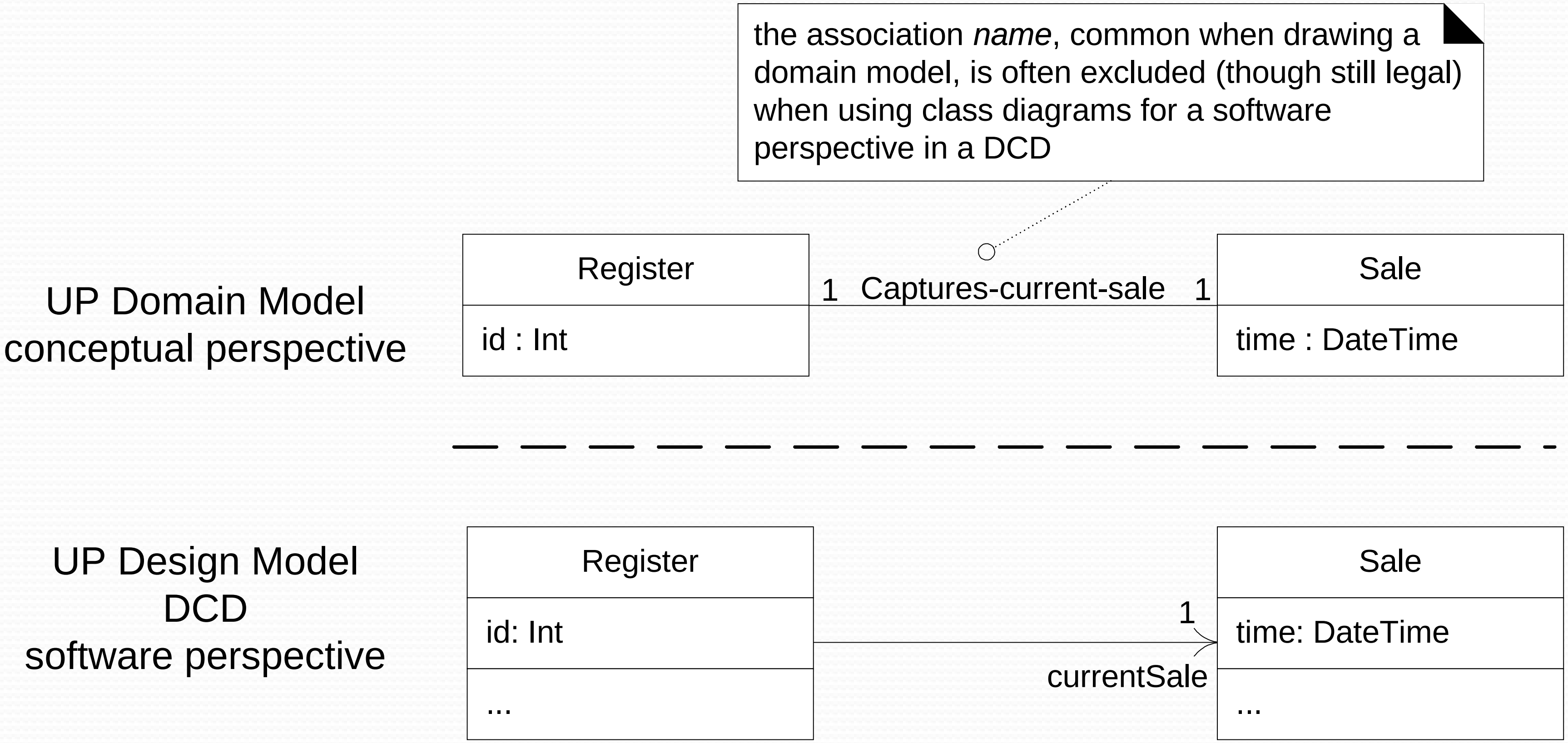
OBSERVE: this style *visually* emphasizes the connection between these classes

using the association notation to indicate Register has a reference to one Sale instance

thorough and unambiguous, but some people dislike the possible redundancy

Larman Fig 16.3

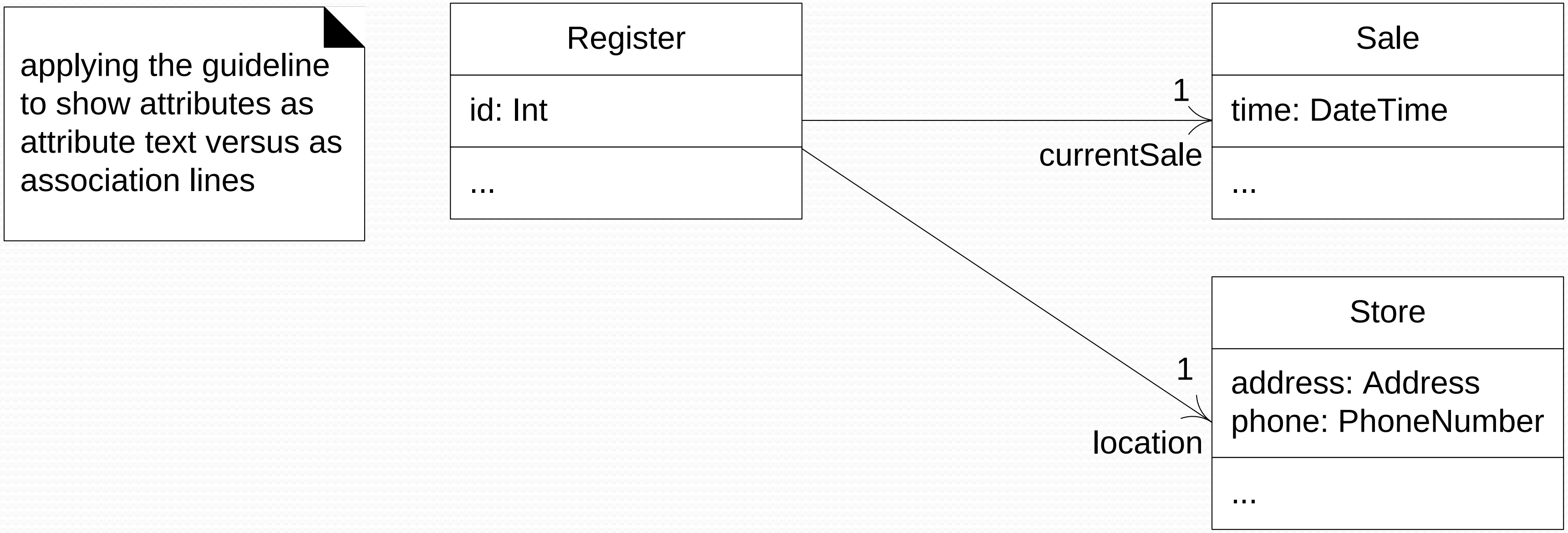
DCD - Attributes And Association



- navigability arrow
- multiplicity @ target end
- rolename at target end to show attribute name

Larman Fig 16.4

DCD - Attributes And Association



applying the guideline to show attributes as attribute text versus as association lines

Register

id: Int

...

Sale

time: DateTime

...

Store

address: Address
phone: PhoneNumber

...

1

currentSale

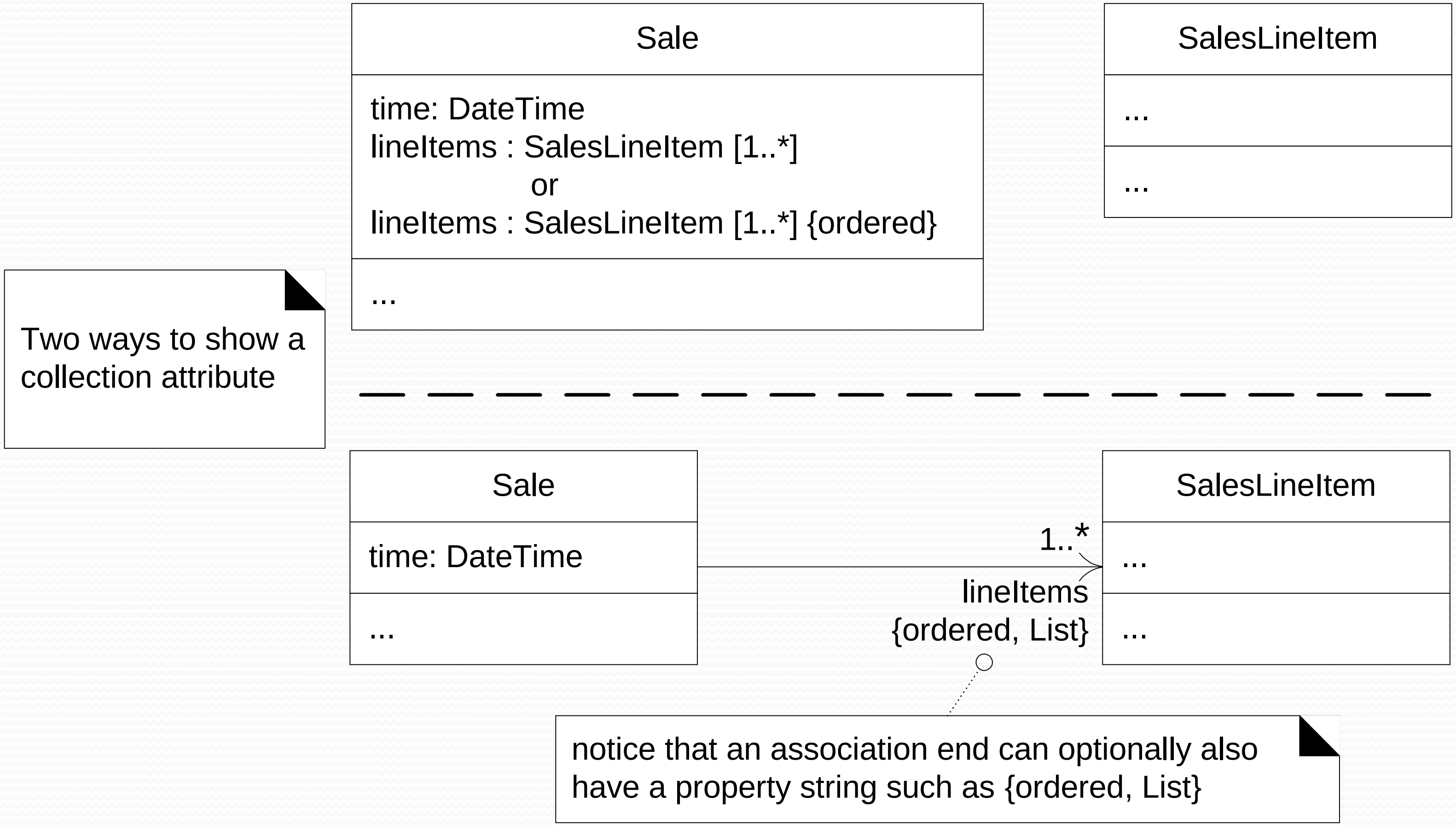
1

location

What are the attributes for the Register class?

Larman Fig 16.5

DCD - Attributes And Association


Sale
time: DateTime
lineItems : SalesLineItem [1..*]
or
lineItems : SalesLineItem [1..*] {ordered}
...
SalesLineItem
...
...
Two ways to show a collection attribute
Sale
time: DateTime
...
1..*
lineItems
{ordered, List}
SalesLineItem
...
...
notice that an association end can optionally also have a property string such as {ordered, List}

Ordered and list are UML-defined keywords

Larman Fig 16.6

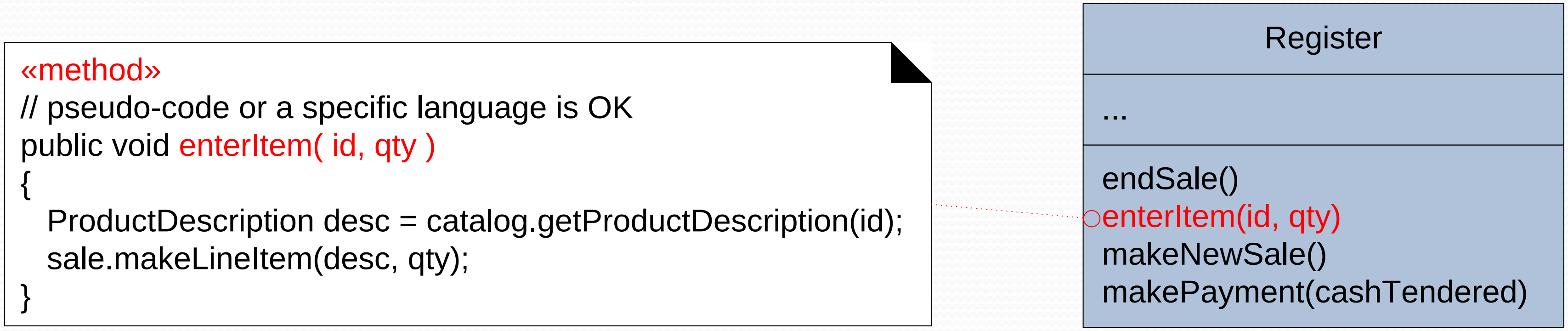
Main Steps in Developing a DCD

- Identify the classes:
  - nouns in the Use Cases Lab 2
  - scan the Conceptual Class Diagram (Domain Model) Lab 3
  - scan the Interaction Diagrams Lab 4
  - list out classes mentioned & those that appear needed:
    - controllers
    - database classes
    - parent classes for classes with a common heritage
    - etc…
- Draw the class diagram

Main Steps in Developing a DCD (2)

- Identify the methods & add them to the DCD:
  - verbs in the Use Cases
  - UML Operation is a declaration with name, parameters, return type, exceptions list and possibly a set of constraints of pre and post conditions
  - Operation Contracts explore the definition of the constraints for UML Operations
  - UML Method is the implementation of an UML Operation

DCD – Showing Method Body



```
«method»
// pseudo-code or a specific language is OK
public void enterItem( id, qty )
{
   ProductDescription desc = catalog.getProductDescription(id);
   sale.makeLineItem(desc, qty);
}
```

Register

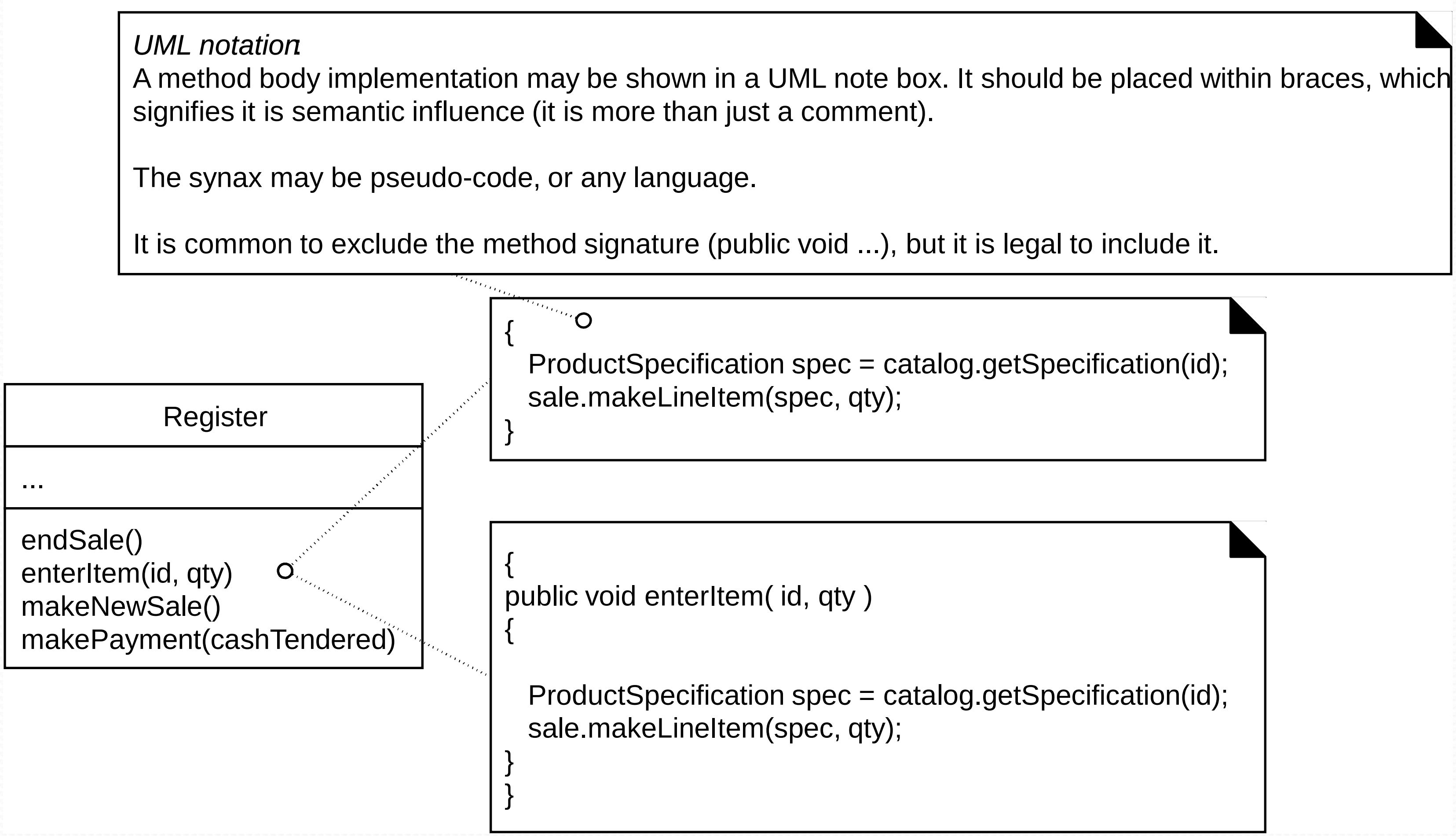
...

endSale()
enterItem(id, qty)
makeNewSale()
makePayment(cashTendered)

- Example:
    - A method illustration in the DCD using a UML note symbol stereotyped with <<method>>

Larman Fig 16.7

Showing Method Body



UML notation:
A method body implementation may be shown in a UML note box. It should be placed within braces, which signifies it is semantic influence (it is more than just a comment).

The synax may be pseudo-code, or any language.

It is common to exclude the method signature (public void ...), but it is legal to include it.

```
{
   ProductSpecification spec = catalog.getSpecification(id);
   sale.makeLineItem(spec, qty);
}
```

Register

...

endSale()
enterItem(id, qty)
makeNewSale()
makePayment(cashTendered)

```
{
public void enterItem( id, qty )
{

   ProductSpecification spec = catalog.getSpecification(id);
   sale.makeLineItem(spec, qty);
}
}
```

DCD – Method

- Method name issues:
  - interpretation of the `create()` message
    - common to omit create methods from DCDs - programming language inconsistencies
- Note that *create* is not a method for each class – it is a concept used to make an object appear based on its class
  - In C++ and Java, the *new* operation actually implements the *create* concept

DCD – Method

- depiction of accessing methods
  - commonly omitted as well

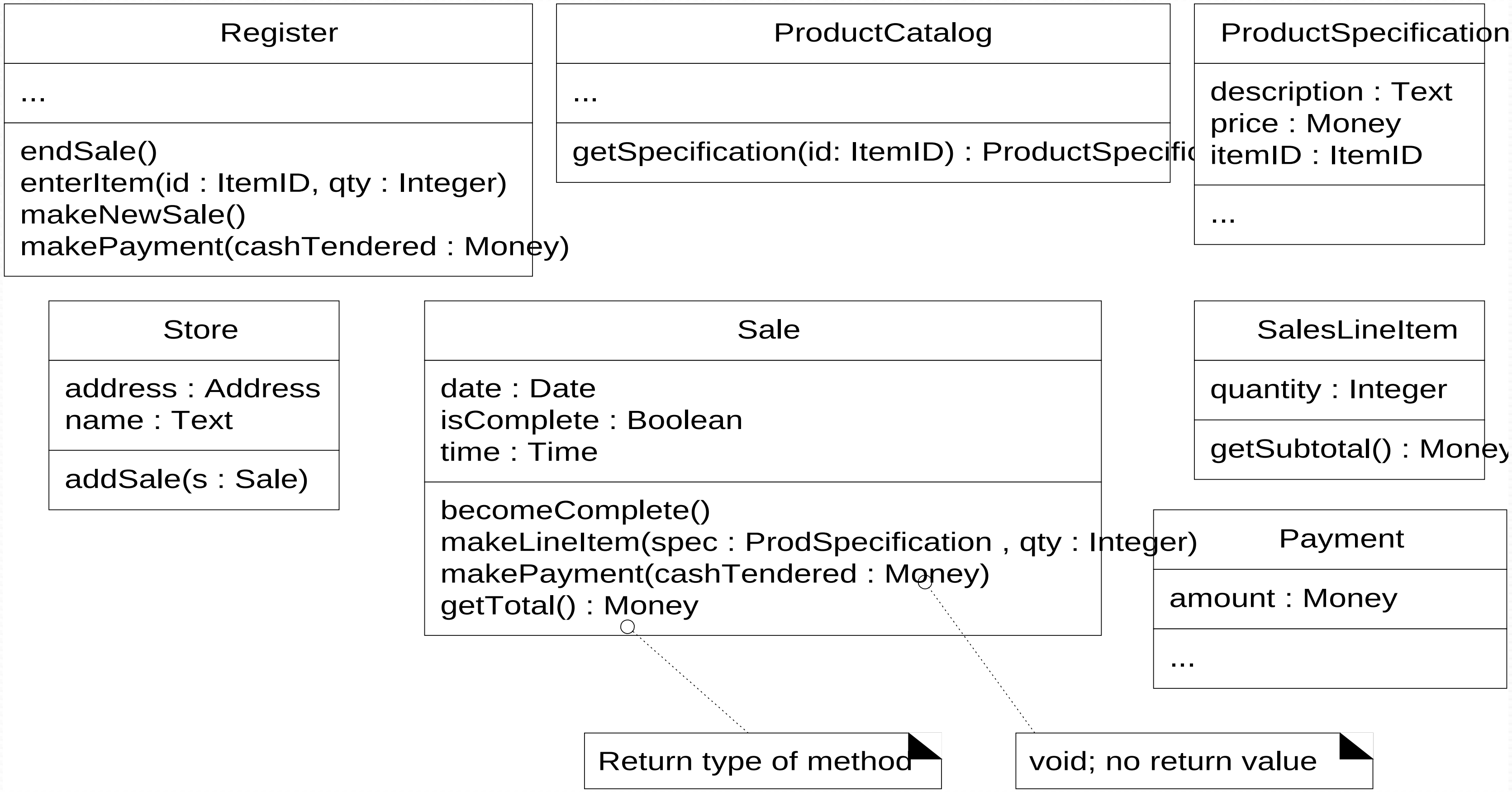
- *Getters* and *Setters* (formally known as accessor and mutator methods, or accessing methods) do <u>not</u> have to be shown on DCD

- interpretation of messages to multi-objects
  - language-dependent syntax

Adding More Type Information

- Consider the audience to determine level of detail:
  - for a CASE tool with code generation, you will want all the details
  - for developers to read and discuss, you will want to suppress the routine information that might clutter the message:
    - get/set methods
    - create() methods

Type Information in the POS Application



Register

...

endSale()
enterItem(id : ItemID, qty : Integer)
makeNewSale()
makePayment(cashTendered : Money)

ProductCatalog

...

getSpecification(id: ItemID) : ProductSpecifi

ProductSpecification

description : Text
price : Money
itemID : ItemID

...

Store

address : Address
name : Text

addSale(s : Sale)

Sale

date : Date
isComplete : Boolean
time : Time

becomeComplete()
makeLineItem(spec : ProdSpecification , qty : Integer)
makePayment(cashTendered : Money)
getTotal() : Money

SalesLineItem

quantity : Integer

getSubtotal() : Money

Payment

amount : Money

...

Return type of method

void; no return value

DCD Associations

- In a DCD, associations are only those needed to make the software work
    - Fulfill visibility and memory needs dictated by the interaction diagrams

- In contrast, the domain model could show all possible associations

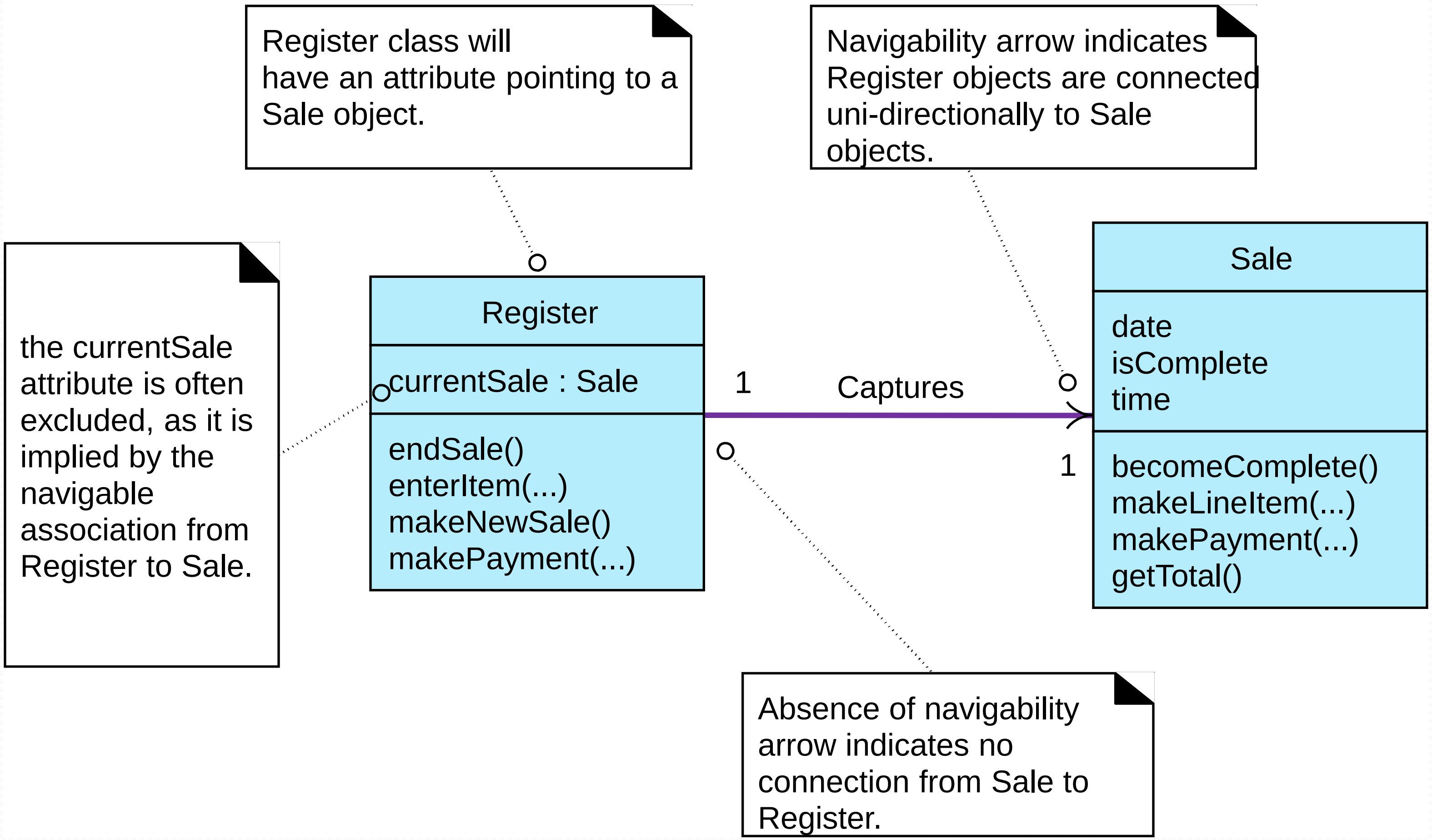
Navigability

- Each association can show whether it is possible to navigate that direction by using an arrowhead

- No arrowheads implies bi-directional navigability

- Navigability implies visibility, usually attribute visibility

Adding Associations and Navigability



Register class will have an attribute pointing to a Sale object.

Navigability arrow indicates Register objects are connected uni-directionally to Sale objects.

the currentSale attribute is often excluded, as it is implied by the navigable association from Register to Sale.

Register

currentSale : Sale

endSale()
enterItem(...)
makeNewSale()
makePayment(...)

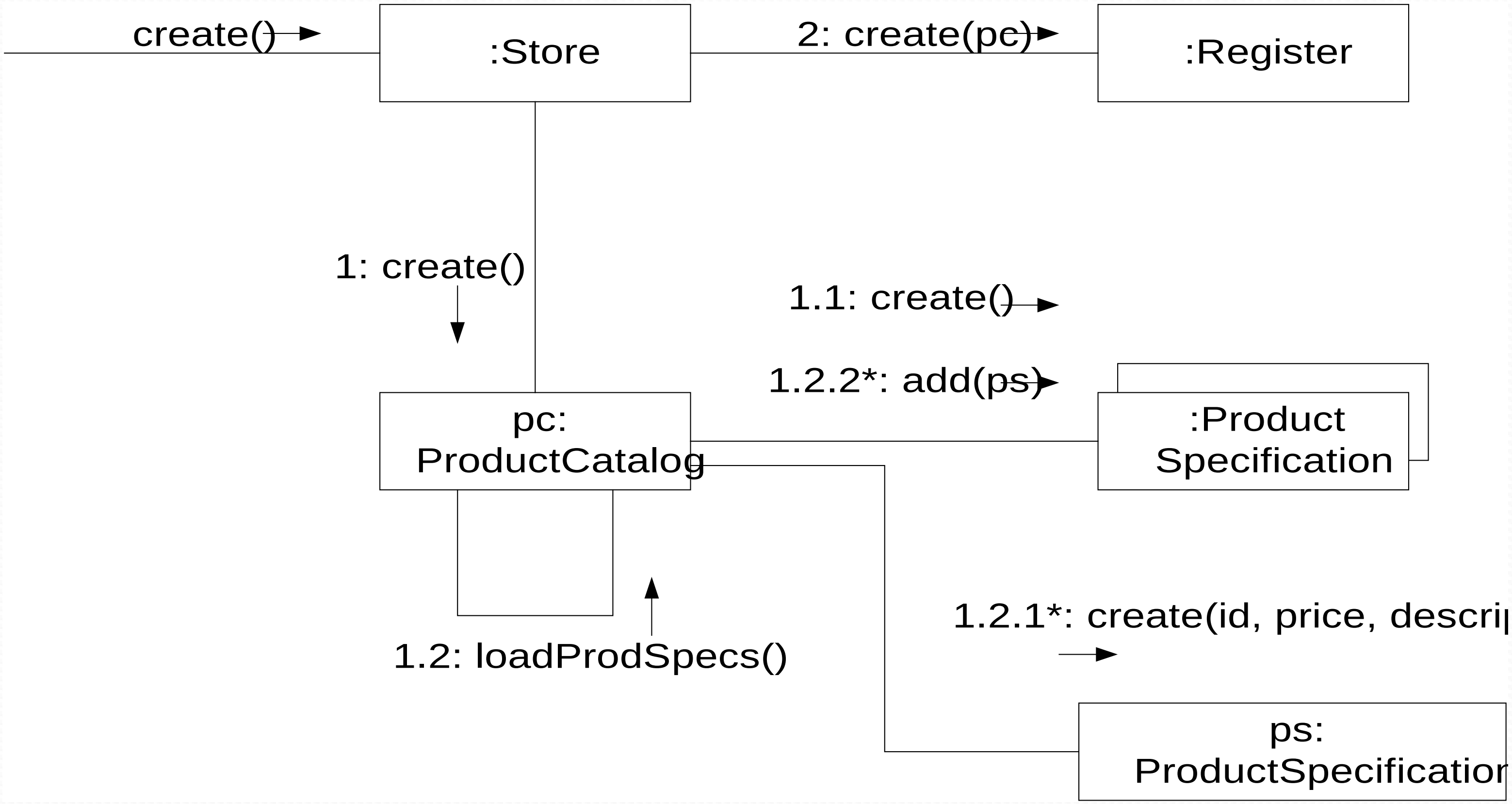
Sale

date
isComplete
time

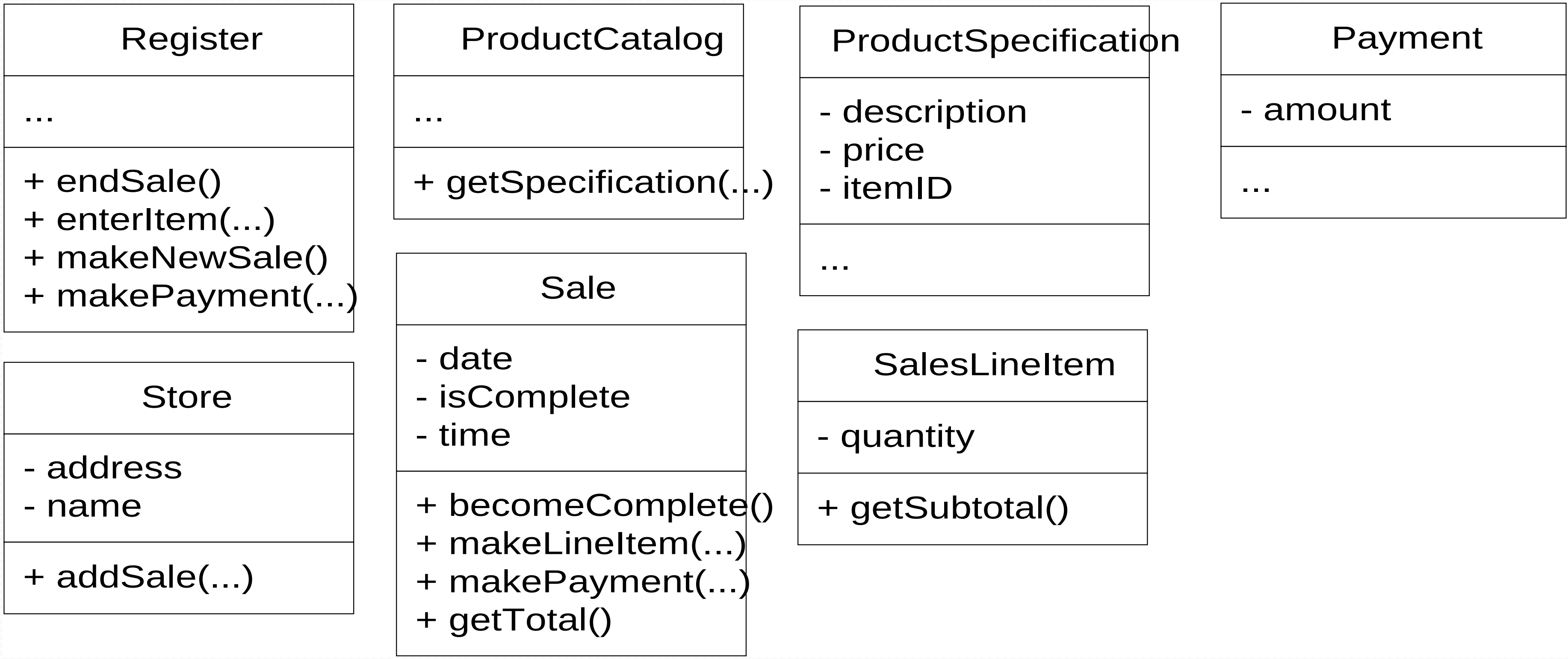
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()

Absence of navigability arrow indicates no connection from Sale to Register.

Navigability is Defined from the Interaction Diagrams



Example – Ready To Add Associations



Register

…

+ endSale()
+ enterItem(…)
+ makeNewSale()
+ makePayment(…)

Store

- address
- name

+ addSale(…)

ProductCatalog

…

+ getSpecification(…)

Sale

- date
- isComplete
- time

+ becomeComplete()
+ makeLineItem(…)
+ makePayment(…)
+ getTotal()

ProductSpecification

- description
- price
- itemID

…

SalesLineItem

- quantity

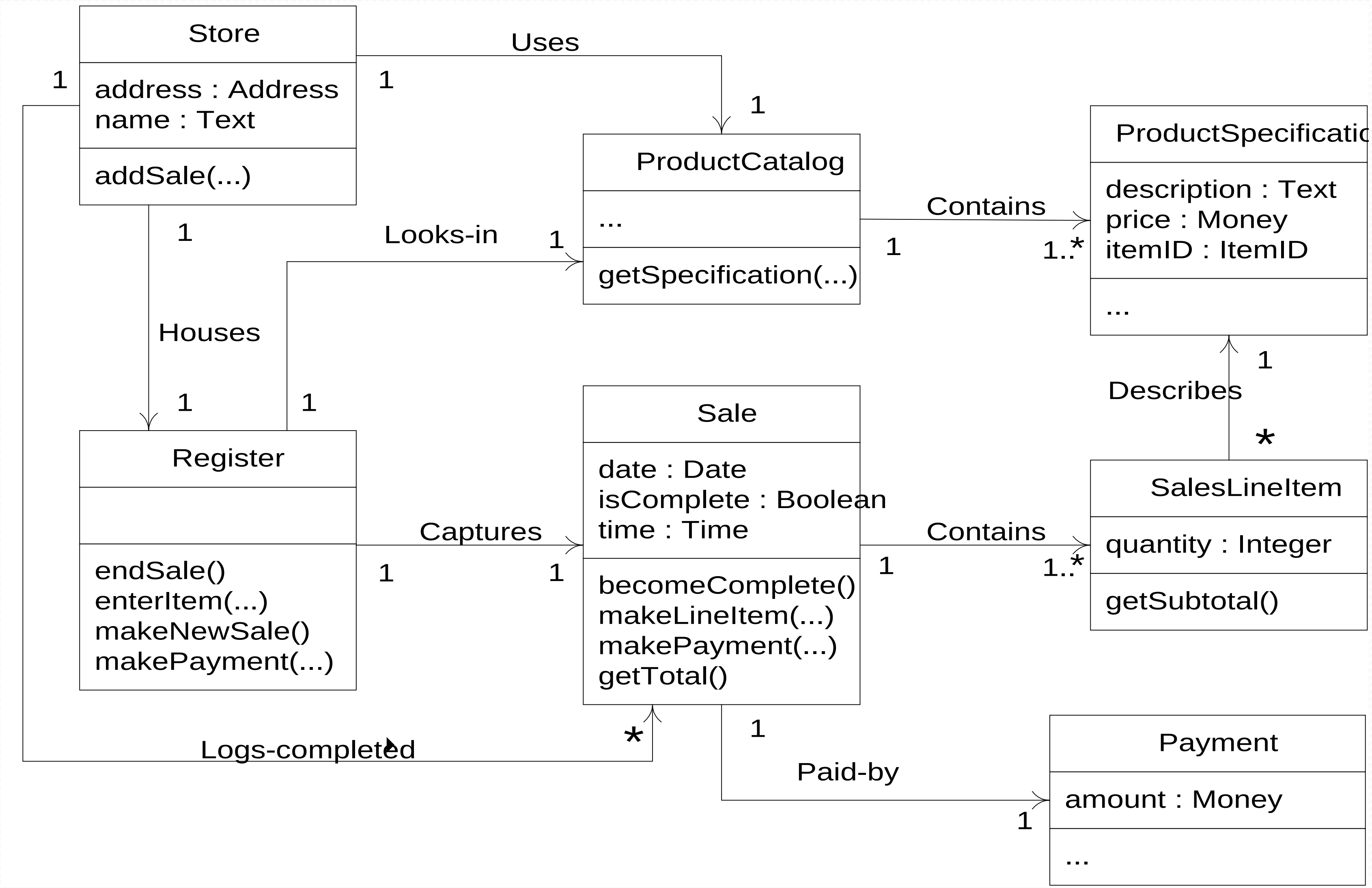
+ getSubtotal()

Payment

- amount

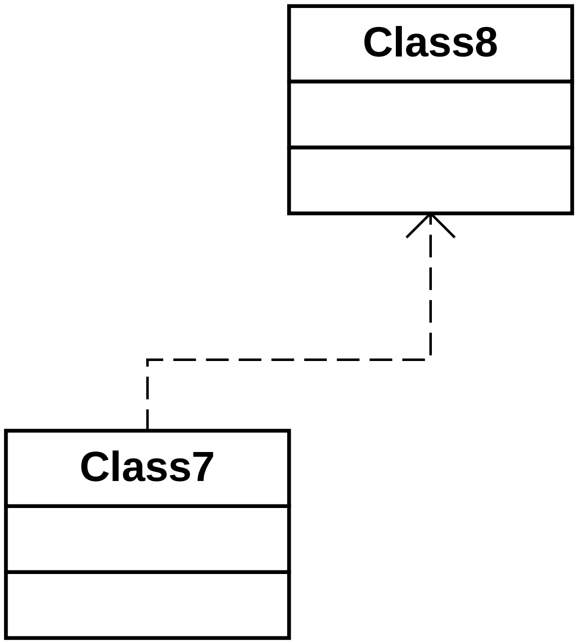
…

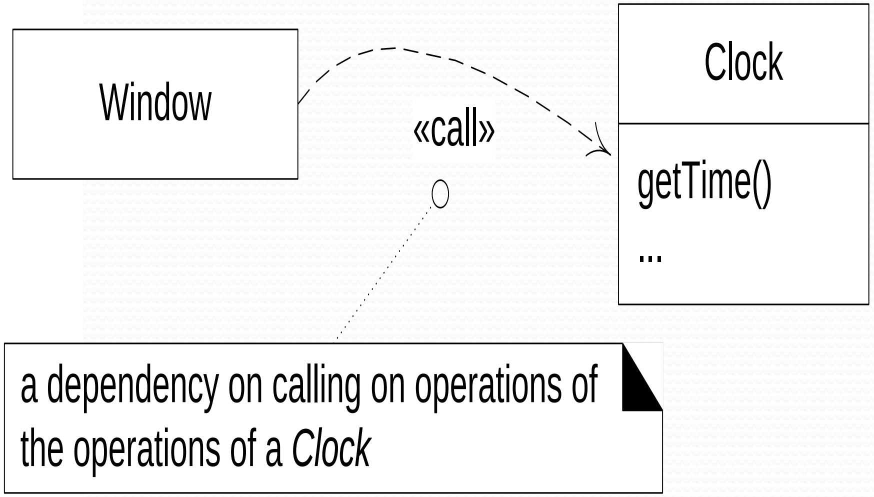
Example: Associations AND Navigability



Dependency Relationships

- A dashed line with an arrow at the end is used to show a dependency relationship and to show non-attribute visibility between classes



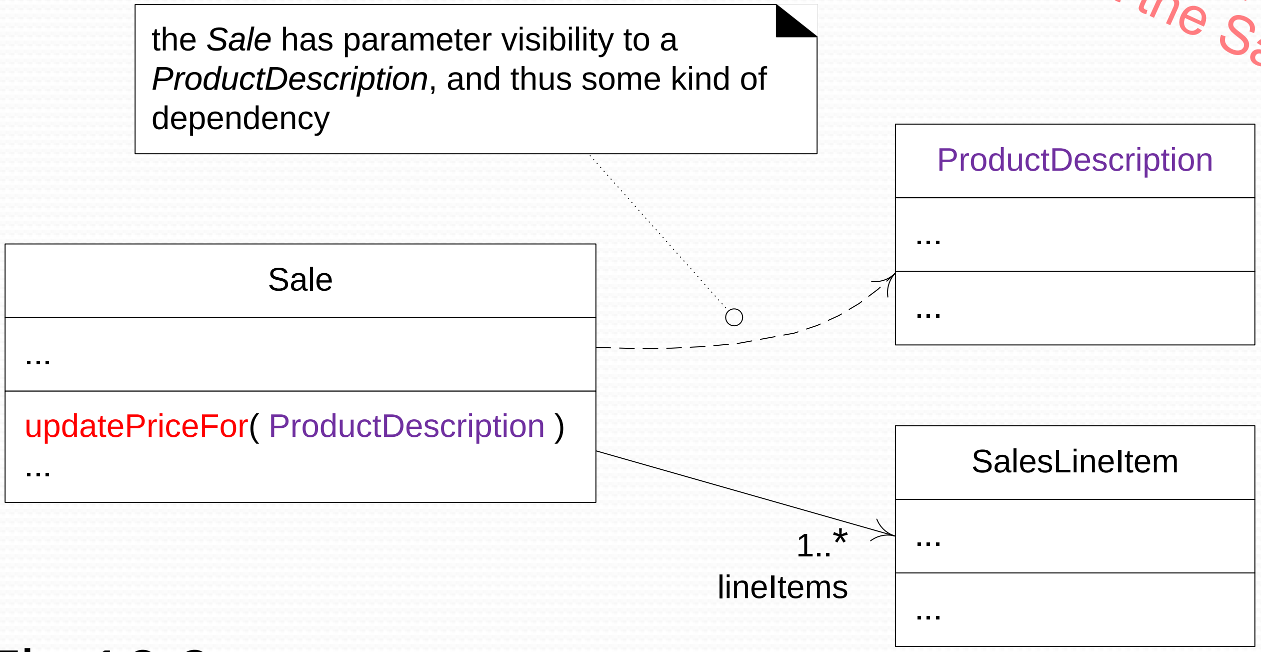


a dependency on calling on operations of the operations of a *Clock*

Dependency At Work

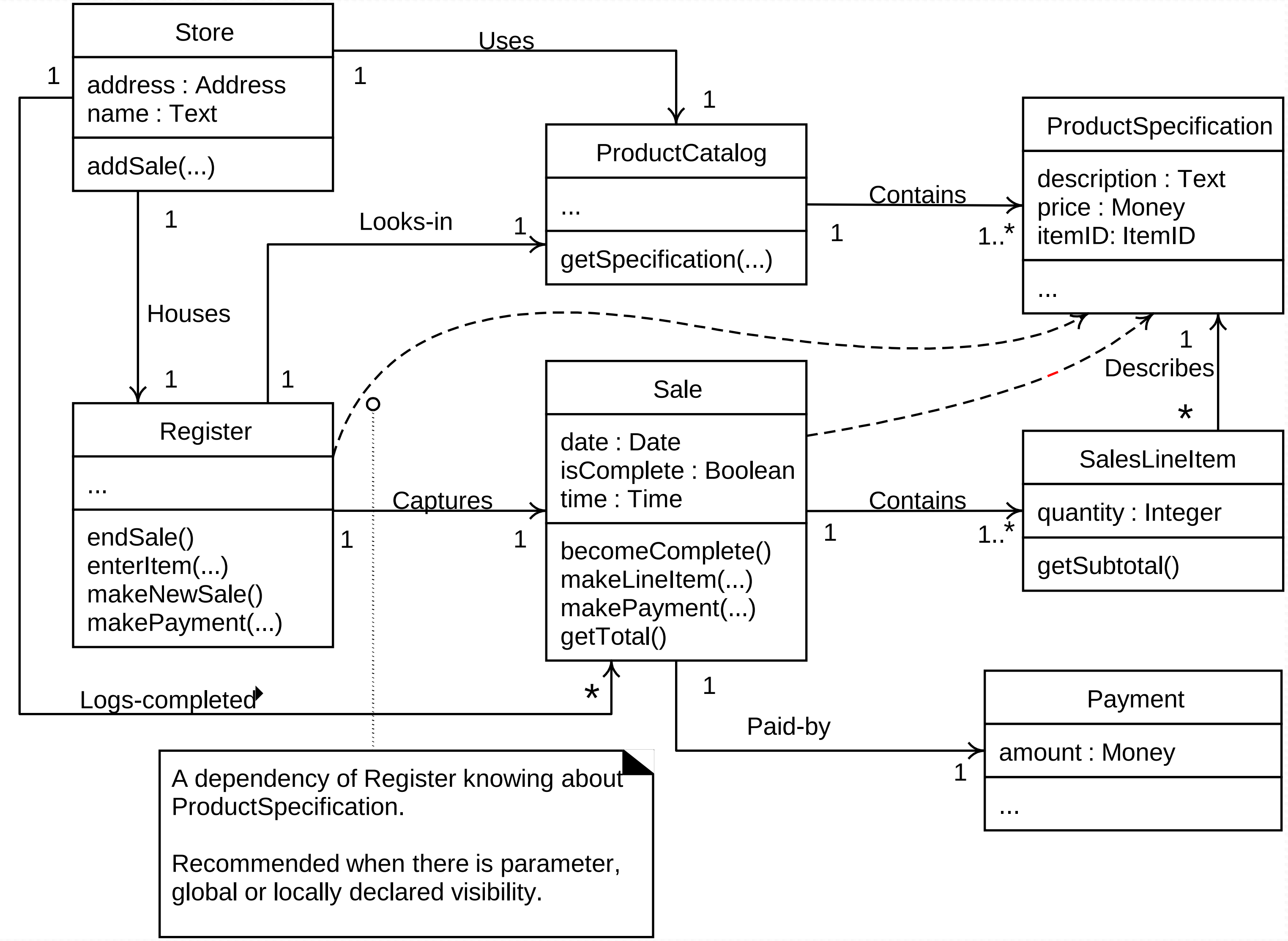
```
Public class Sale
      {
      Public void updatePriceFor (ProductDescription description)
      {
          Money basePrice = description .getPrice();
          // ..
      }}
```

Changing the ProductDescription class can affect the Sale class


the *Sale* has parameter visibility to a *ProductDescription*, and thus some kind of dependency

Larman Fig 16.9

Adding Dependency Relationships

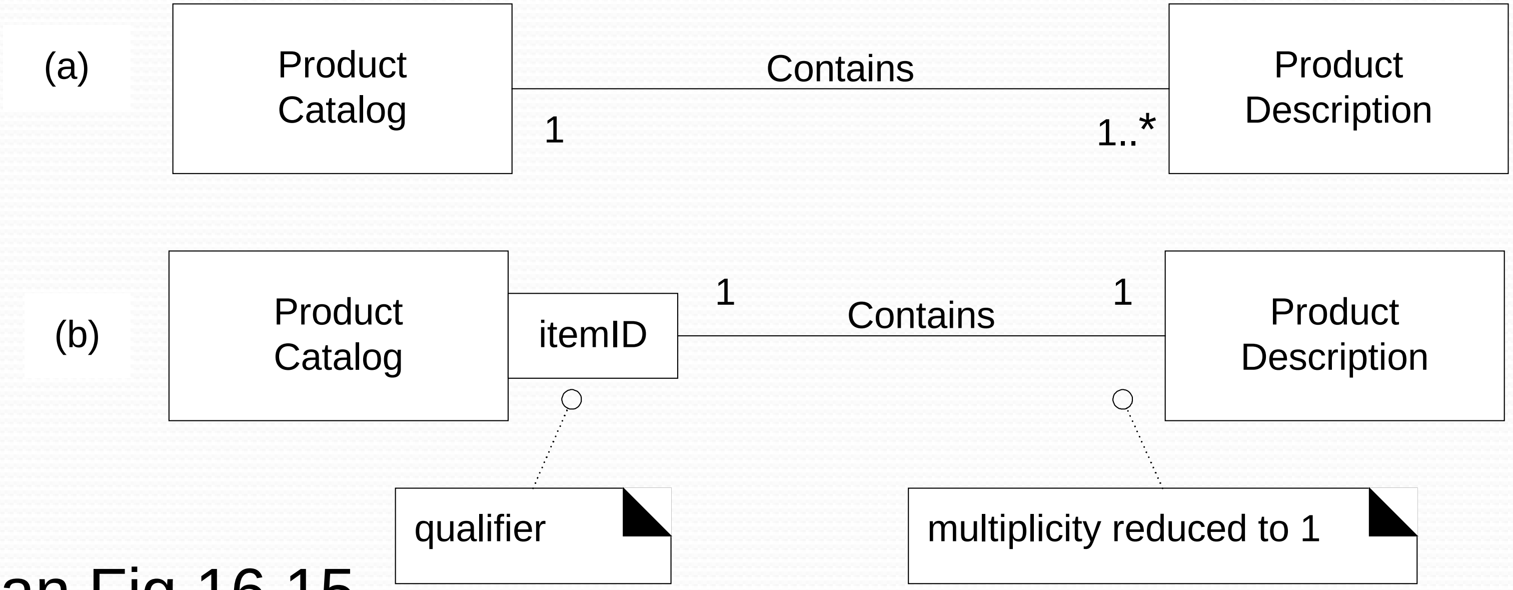


A dependency of Register knowing about ProductSpecification.

Recommended when there is parameter, global or locally declared visibility.

Qualified Association

- A *qualifier* is added to a *qualified association*
- It is used to select an object from a larger set of related objects based on the *qualifier key*
- *qualifier* reduces multiplicity at target end of the association to 1



Larman Fig 16.15

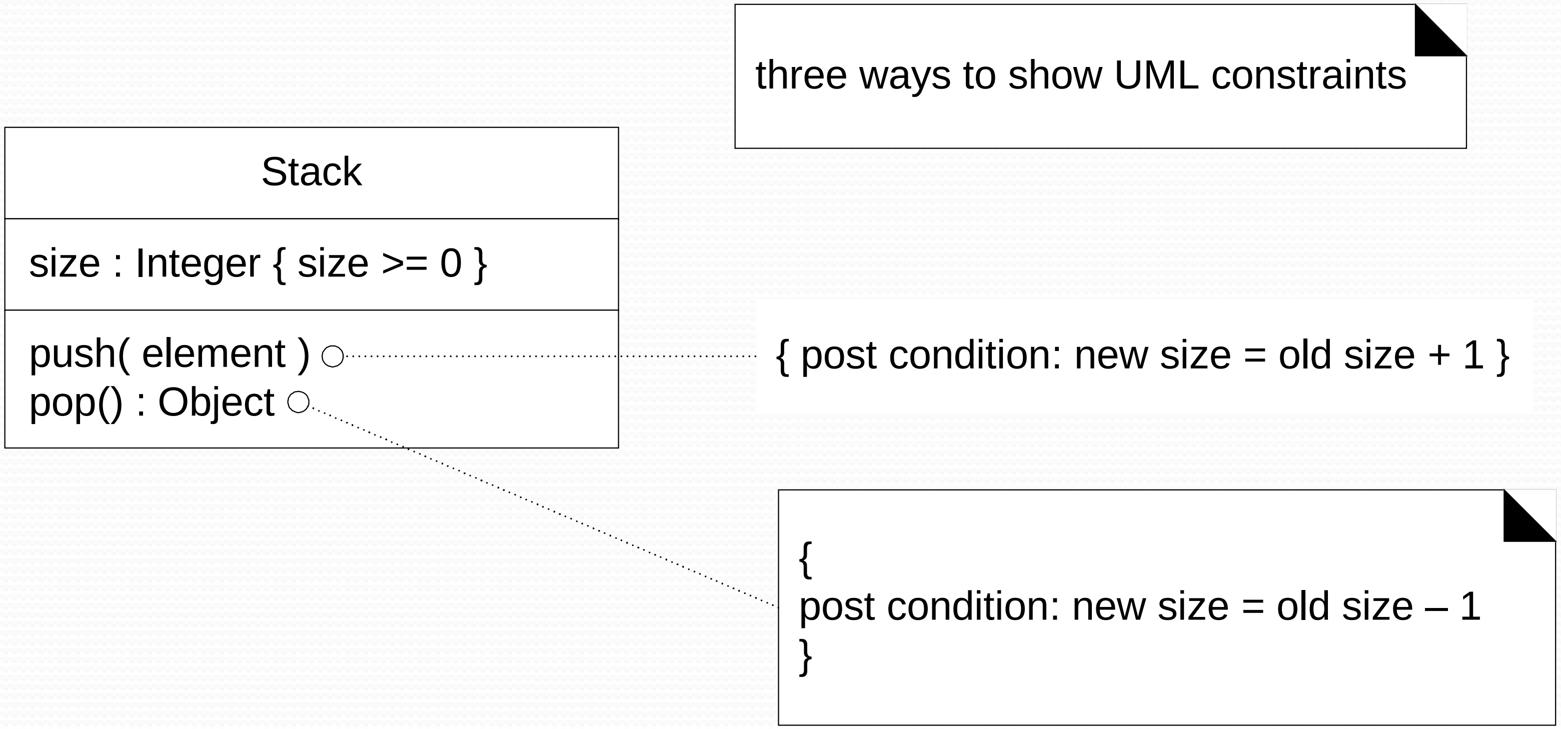
Sample UML Keywords

Keyword	Meaning	Example USage
<<actor>>	classifier is an actor	above classifier name
<<interface>>	classifier is an interface	above classifier name
<<abstract>>	abstract element; can’t be instantiated	after classifier name or operation name
<<ordered>>	set of objects with imposed order	at an association end

Note: Stereotypes and Keywords are shown with guillemets

Constraints (Restrictions)

- Visualized in text form between braces
  - e.g. { age > 25}



three ways to show UML constraints

Stack

size : Integer { size >= 0 }

push(element) ○
pop() : Object ○

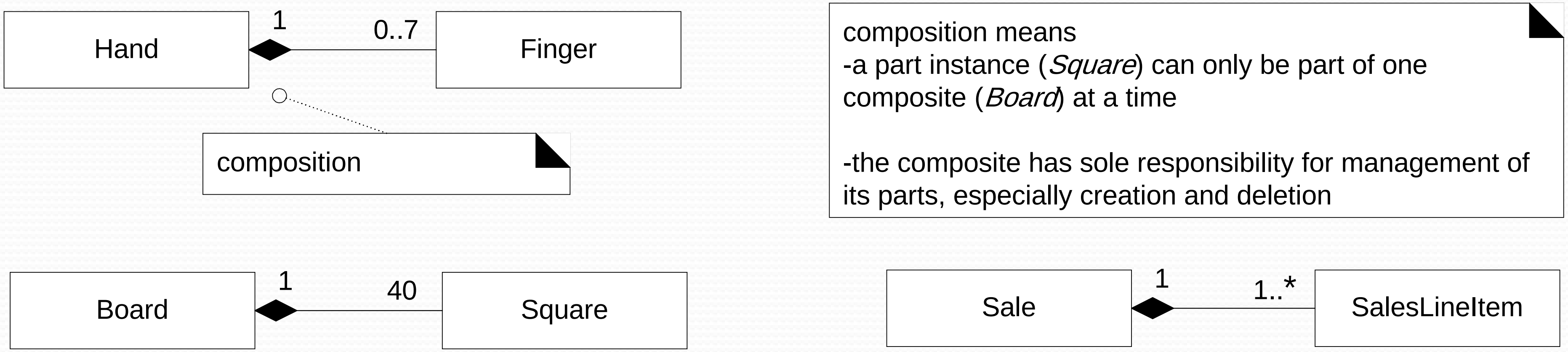
{ post condition: new size = old size + 1 }

{
post condition: new size = old size – 1
}

Larman Fig 16.14

Composition/Aggregation

- The filled-diamond on an association line
- Since “Has-part” is implicit in the association in composition, there is need to name it explicitly



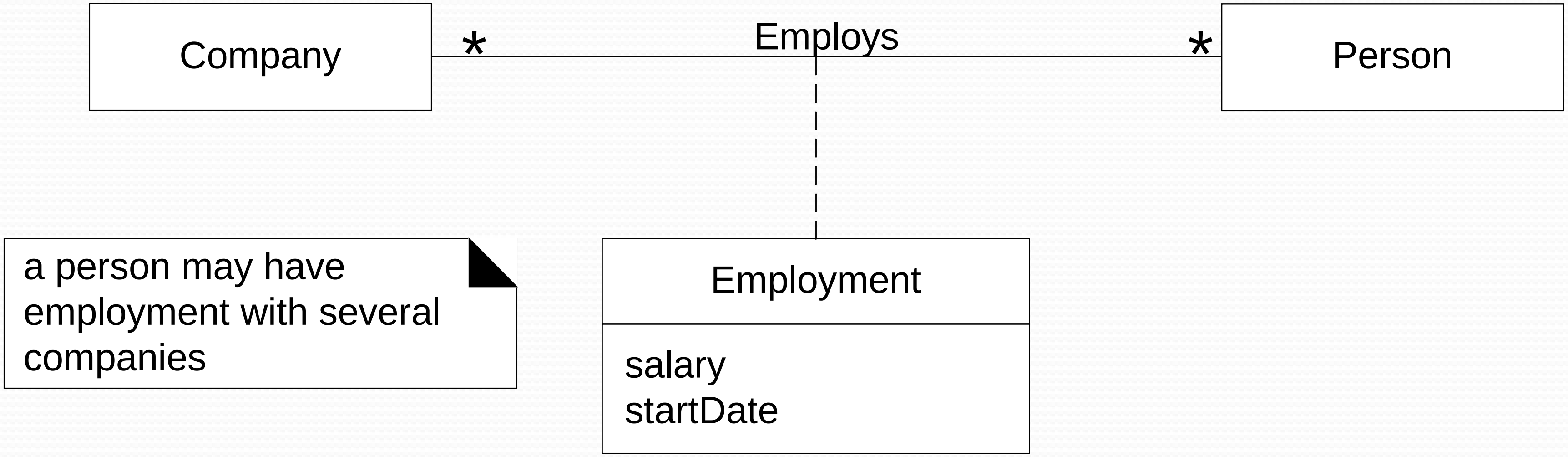
composition means
-a part instance (*Square*) can only be part of one composite (*Board*) at a time

-the composite has sole responsibility for management of its parts, especially creation and deletion

Larman Fig 16.13

Association Class

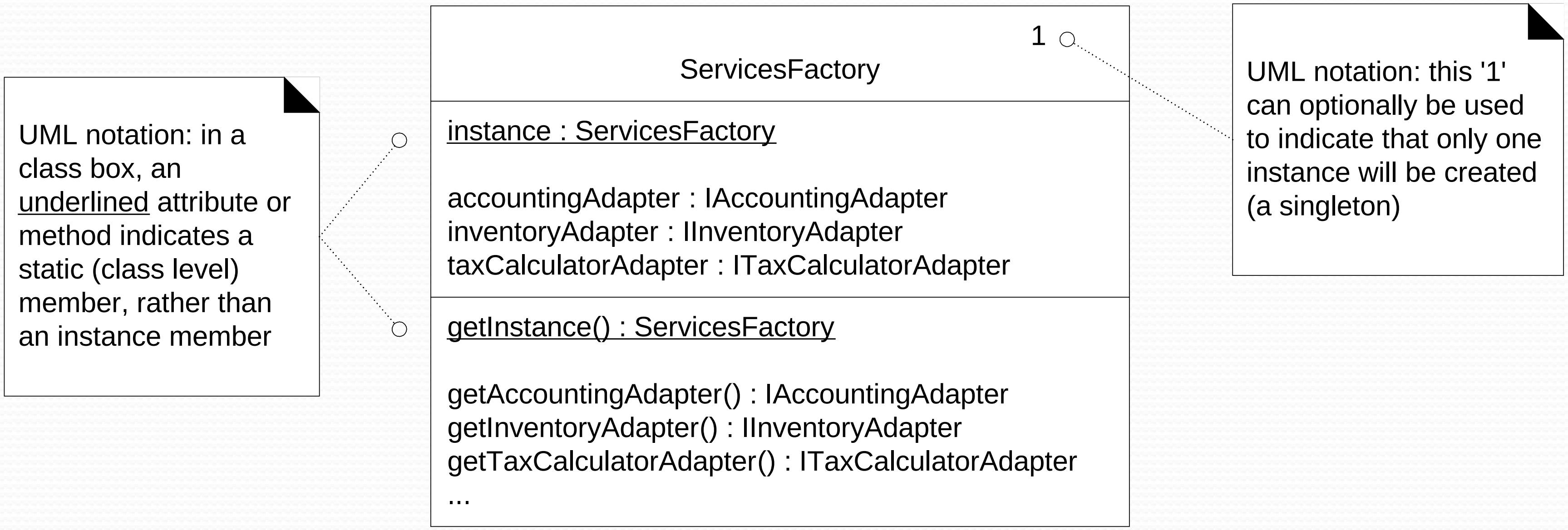
- Treat an association as a class
- Shown by the dashed line from the association to the association class



Larman Fig 16.16

Singleton Class

- One instance of the class is instantiated
- Marked by a 1 in the upper corner o the class name section



UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member

1

ServicesFactory

instance : ServicesFactory

accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter

getInstance() : ServicesFactory

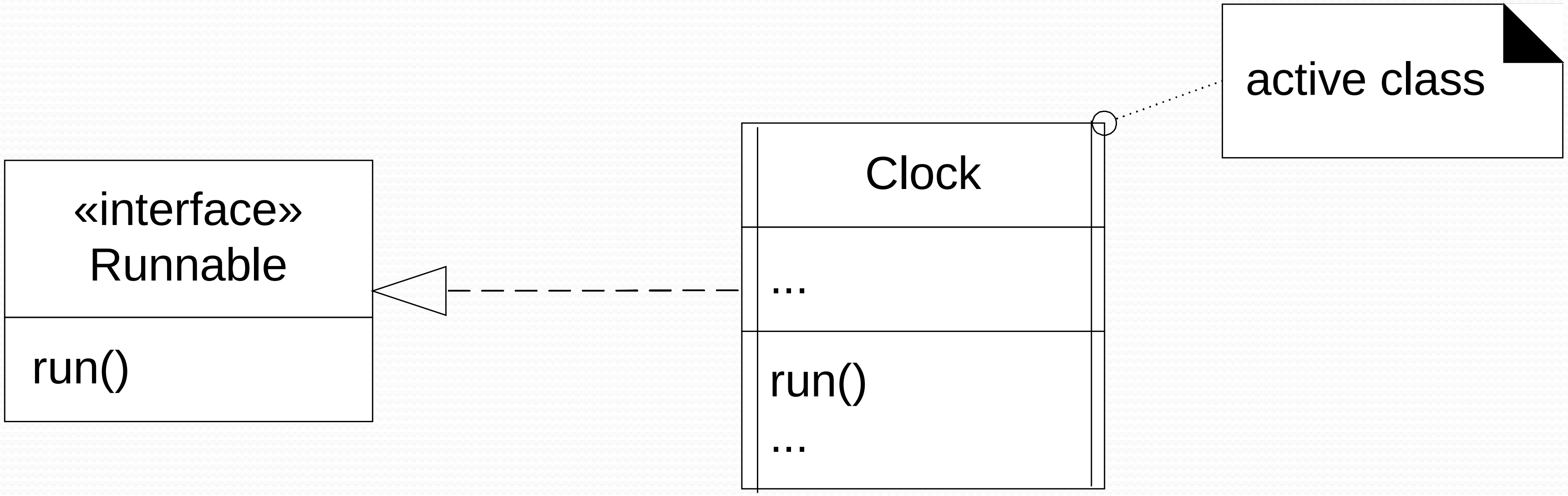
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

Larman Fig 16.17

Active Class

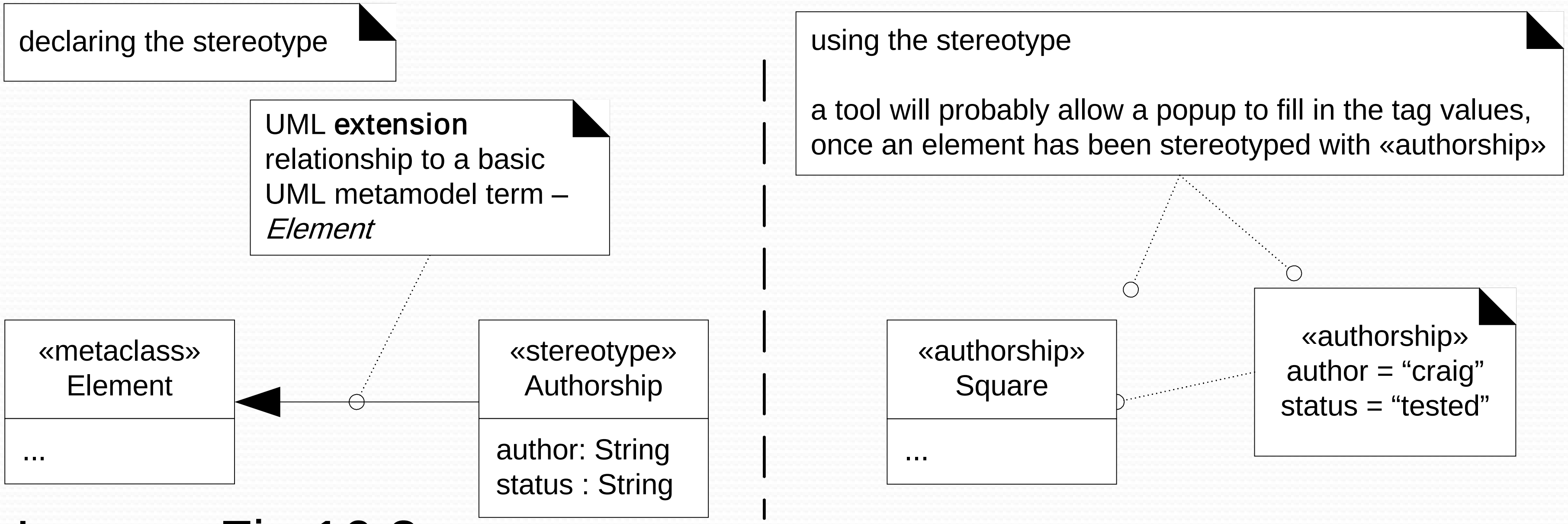
- An active class for active object that controls its own thread of execution
- Marked by double vertical lines on left and right sides of the class box



Larman Fig 16.20

UML - Stereotypes

- UML predefines many stereotypes
  - <<destroy>>
- UML supports user defined stereotypes
- A set of tags can be declared with a stereotype



declaring the stereotype

UML **extension** relationship to a basic UML metamodel term – *Element*

using the stereotype

a tool will probably allow a popup to fill in the tag values, once an element has been stereotyped with «authorship»

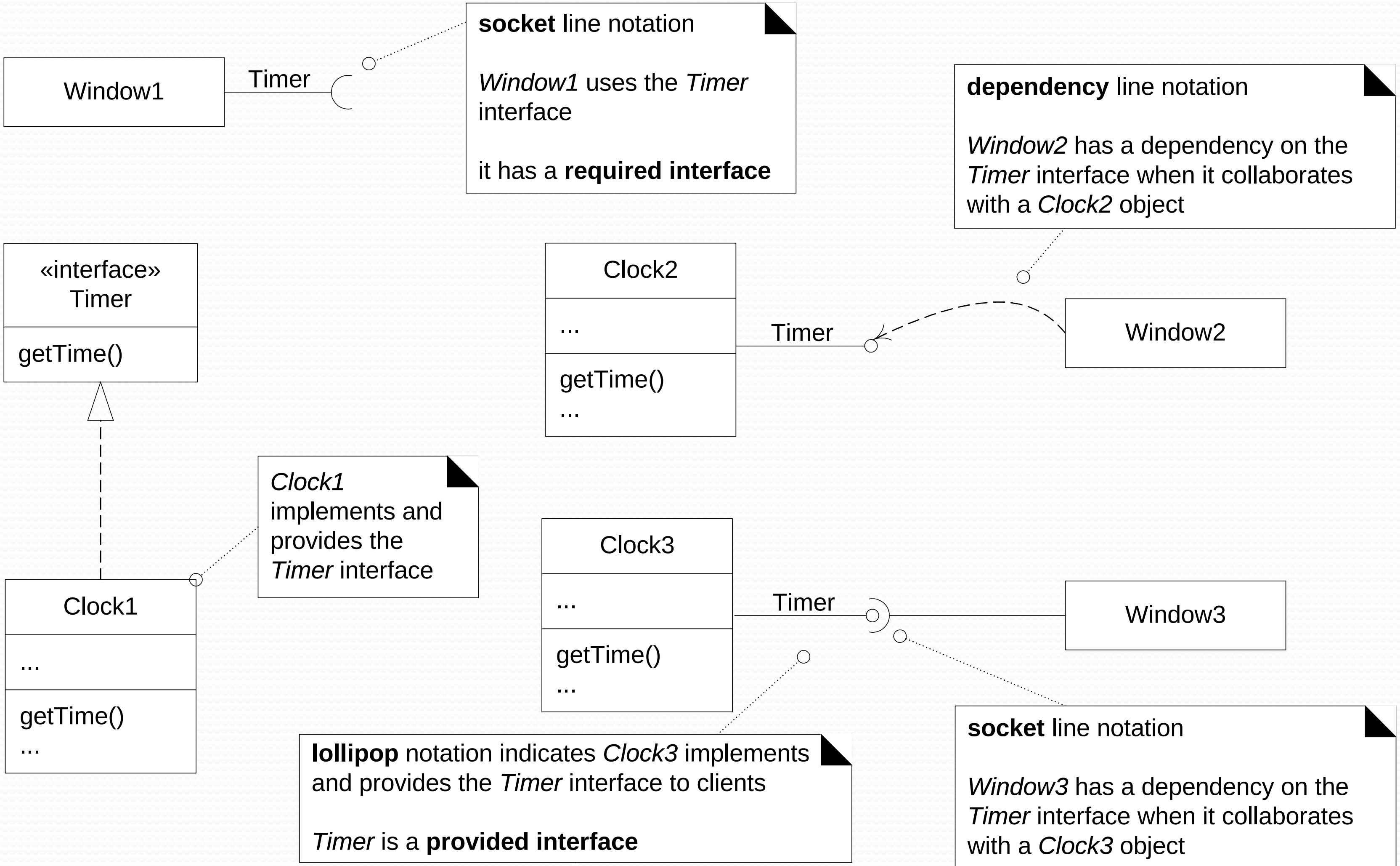
Larman Fig 16.8

UML - Properties

- UML Property is a named value denoting a characteristic of an element
- Usually shown in a Property String
  - e.g. *visibility* is a pre-defined UML property of an operation and may be shown with a value, such as {abstract, *visibility=public*}
- In the case of abstract classes, instead of showing with an {abstract} tag, the class name can be *italicized*

UML - Interface

- Example of a variation of new notations



Window1

Timer

socket line notation

Window1 uses the *Timer* interface

it has a **required interface**

dependency line notation

Window2 has a dependency on the *Timer* interface when it collaborates with a *Clock2* object

«interface»
Timer

getTime()

Clock2

...

getTime()
...

Timer

Window2

Clock1 implements and provides the *Timer* interface

Clock1

...

getTime()
...

Clock3

...

getTime()
...

Timer

Window3

lollipop notation indicates *Clock3* implements and provides the *Timer* interface to clients

Timer is a **provided interface**

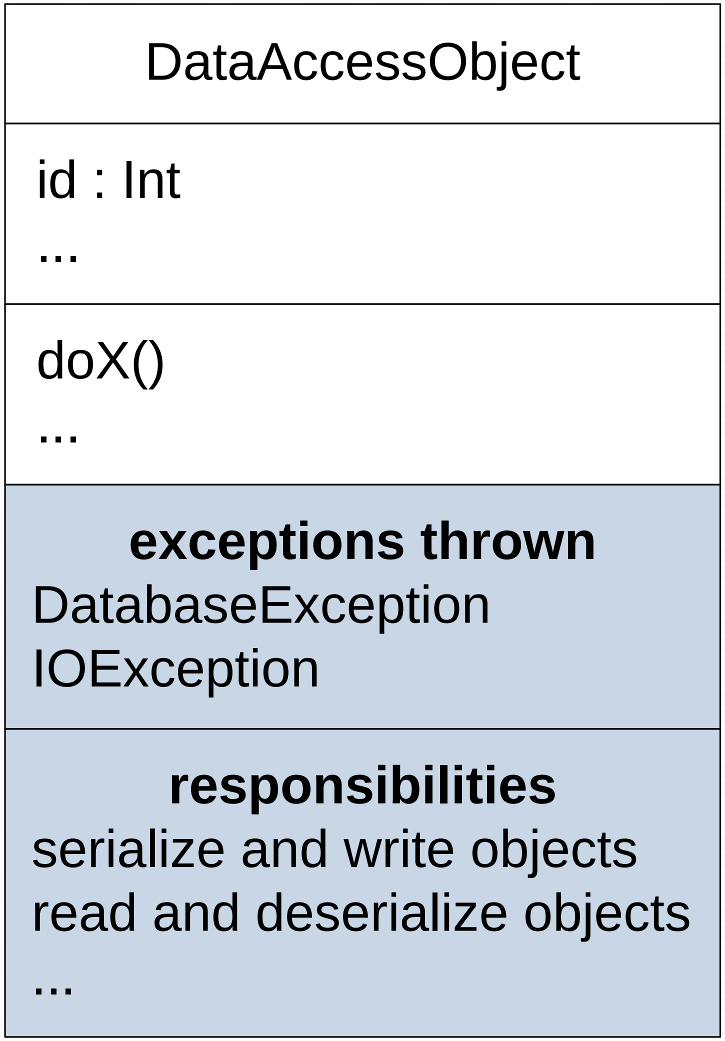
socket line notation

Window3 has a dependency on the *Timer* interface when it collaborates with a *Clock3* object

Larman Fig 16.12

UML – User Defined Section

UML supports additional user defined sections in the class box



DataAccessObject

id : Int
...

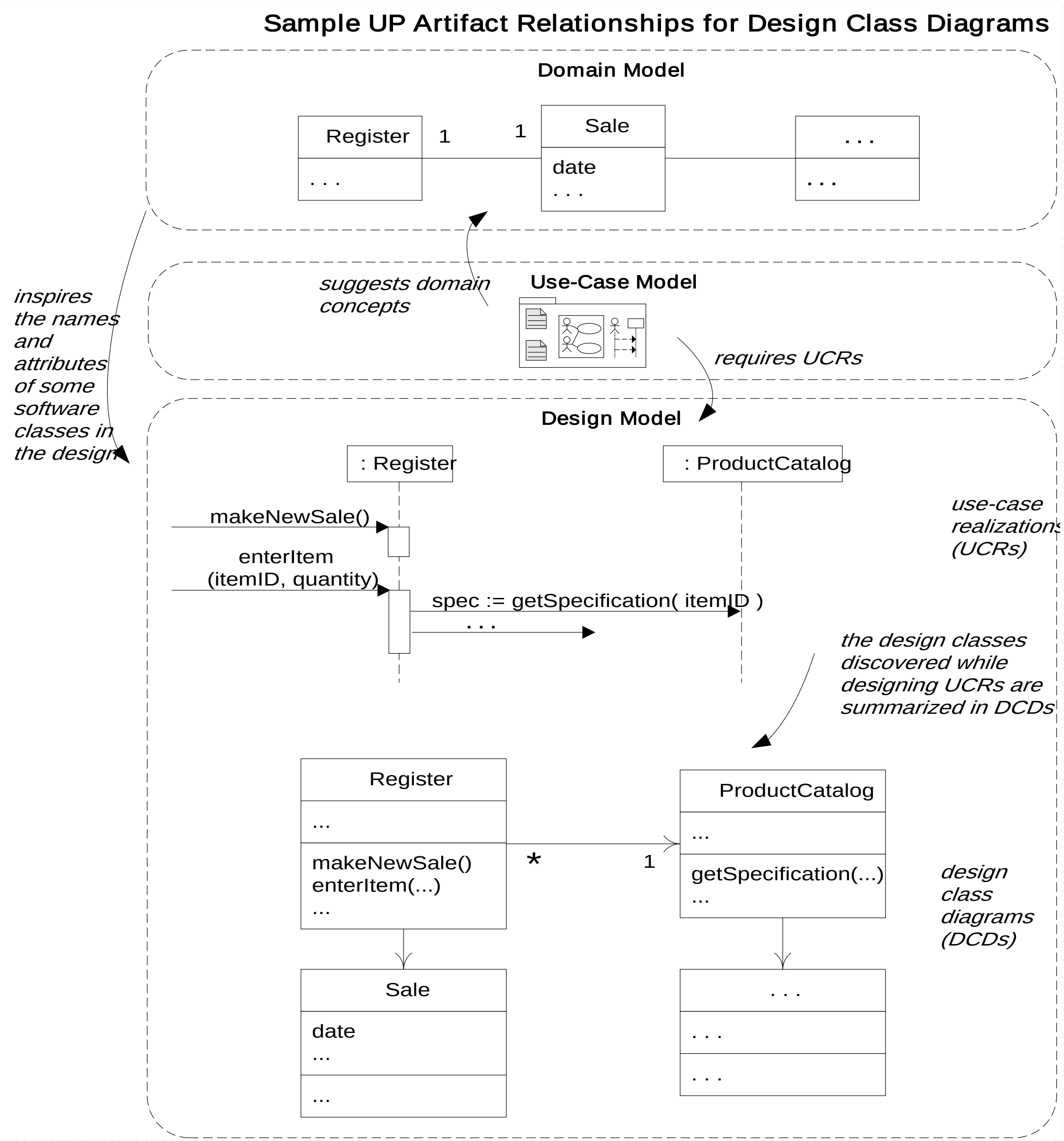
doX()
...

exceptions thrown
DatabaseException
IOException

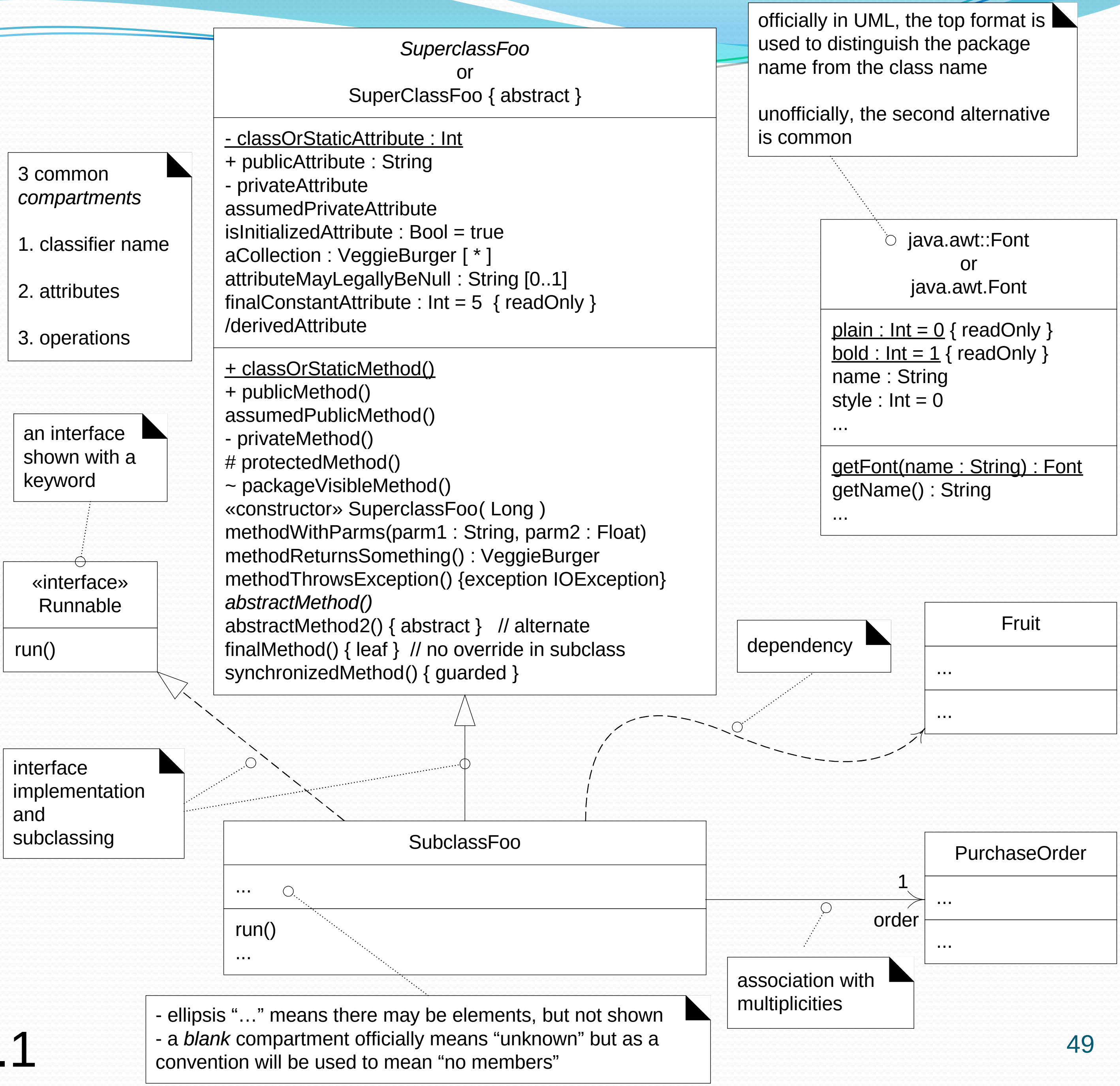
responsibilities
serialize and write objects
read and deserialize objects
...

Larman Fig 16.19

Artifact Relationships


Sample UP Artifact Relationships for Design Class Diagrams
Domain Model
Register
1
1
Sale
date
. . .
. . .
. . .
. . .
suggests domain concepts
Use-Case Model
requires UCRs
inspires the names and attributes of some software classes in the design
Design Model
: Register
: ProductCatalog
makeNewSale()
enterItem (itemID, quantity)
spec := getSpecification(itemID)
. . .
use-case realizations (UCRs)
the design classes discovered while designing UCRs are summarized in DCDs
Register
...
makeNewSale()
enterItem(...)
...
*
1
ProductCatalog
...
getSpecification(...)
...
design class diagrams (DCDs)
Sale
date
...
...
. . .
. . .
. . .

Summary UML Notations



officially in UML, the top format is used to distinguish the package name from the class name

unofficially, the second alternative is common

3 common *compartments*

1. classifier name

2. attributes

3. operations

SuperclassFoo
or
SuperClassFoo { abstract }

- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute

+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

java.awt::Font
or
java.awt.Font

plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...

getFont(name : String) : Font
getName() : String
...

an interface shown with a keyword

«interface»
Runnable

run()

dependency

Fruit

...

...

interface implementation and subclassing

SubclassFoo

...

run()
...

PurchaseOrder

...

...

1

order

association with multiplicities

- ellipsis "…" means there may be elements, but not shown
- a *blank* compartment officially means "unknown" but as a convention will be used to mean "no members"

Larman Fig 16.1

Visibility Defaults

- If visibility is not shown, it means it is not specified in UML notation

- Conventions:
  - assume attributes are private
  - assume methods are public
  - unless otherwise noted…

Summary

- DCD is Design - not a Domain Artifact
- Level of detail shown appropriate to the audience
- DCD is built from existing artifacts:
  - assembling design model may lead you to revise previously generated artifacts, including the domain model
- DCD generally built in Elaboration phase
- DCD is a technical design document, not a user domain artifact