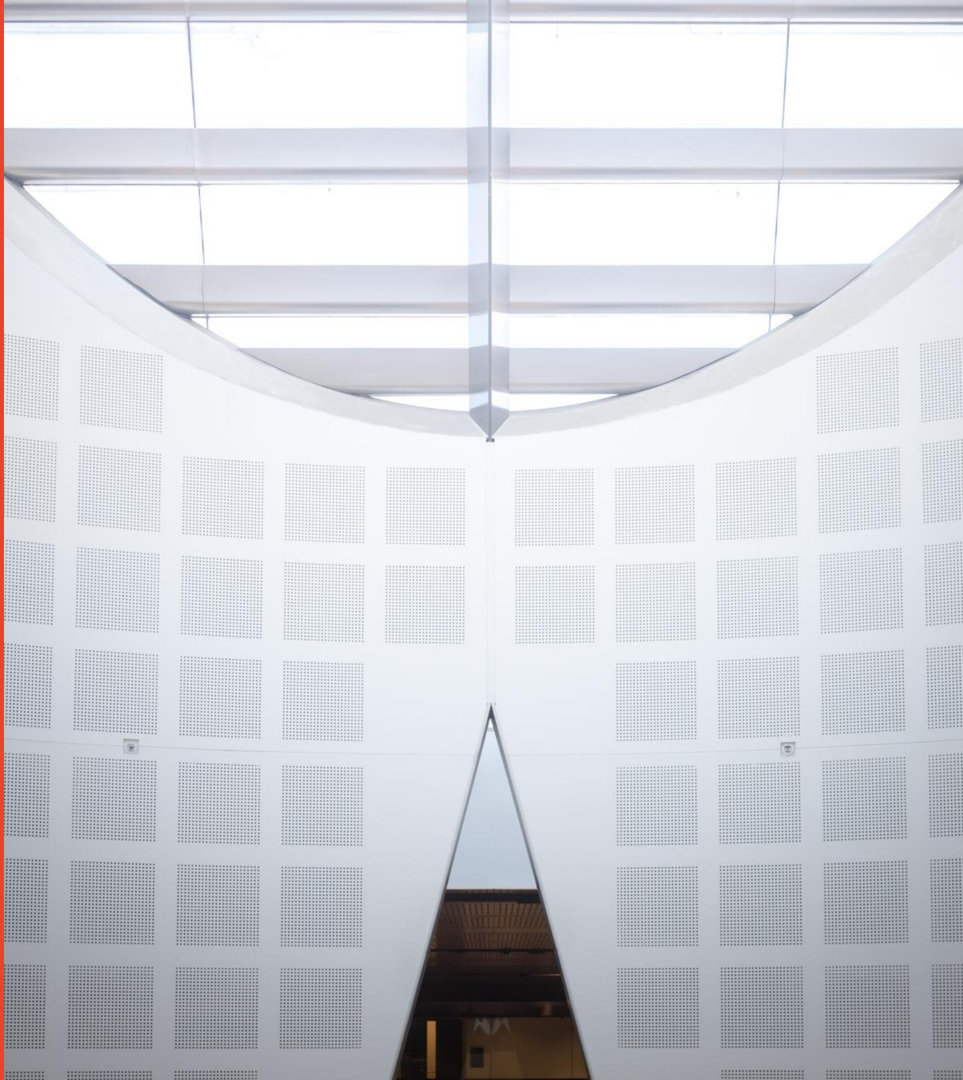
# Software Design and Construction 1 SOFT2201 / COMP9201

## Software Modeling Case Studies (UML Modeling)

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



# Agenda

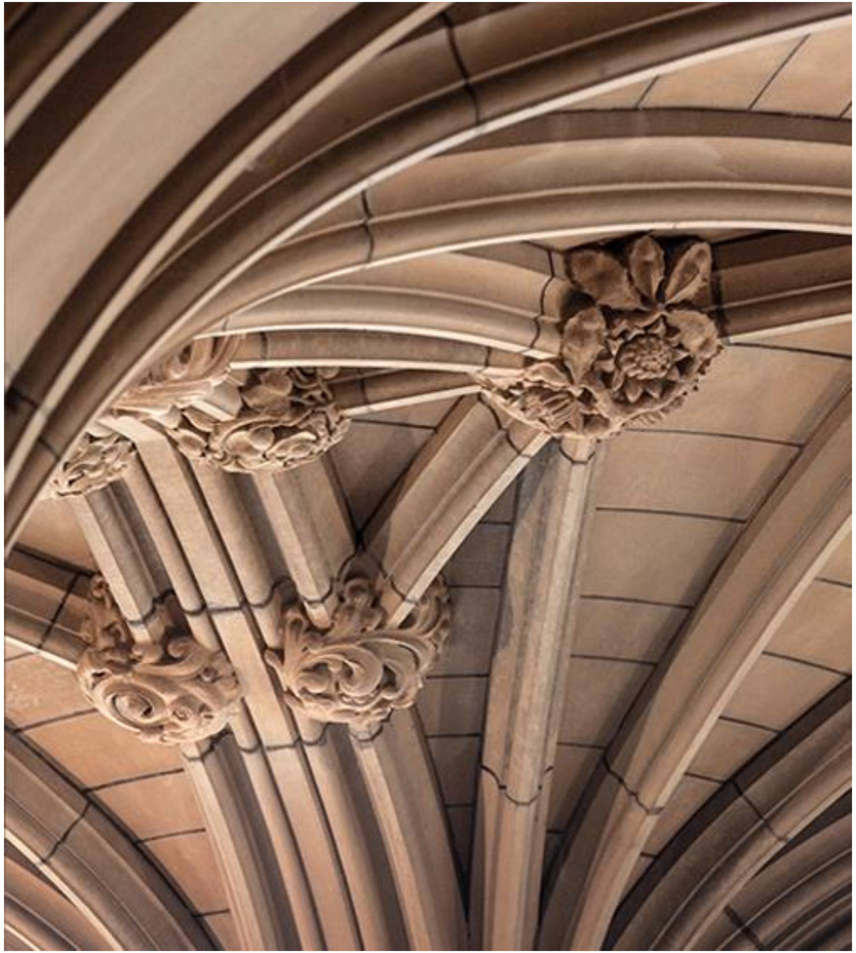
- UML Modeling
  - UML Use Case Diagrams
  - UML Class Diagrams
  - UML Interaction Diagrams
- Case Study
  - Next Gen Point-of-Sale (POS) System
  - Use Cases
  - Domain models
  - Class and Sequence Diagrams

# UML Modelling

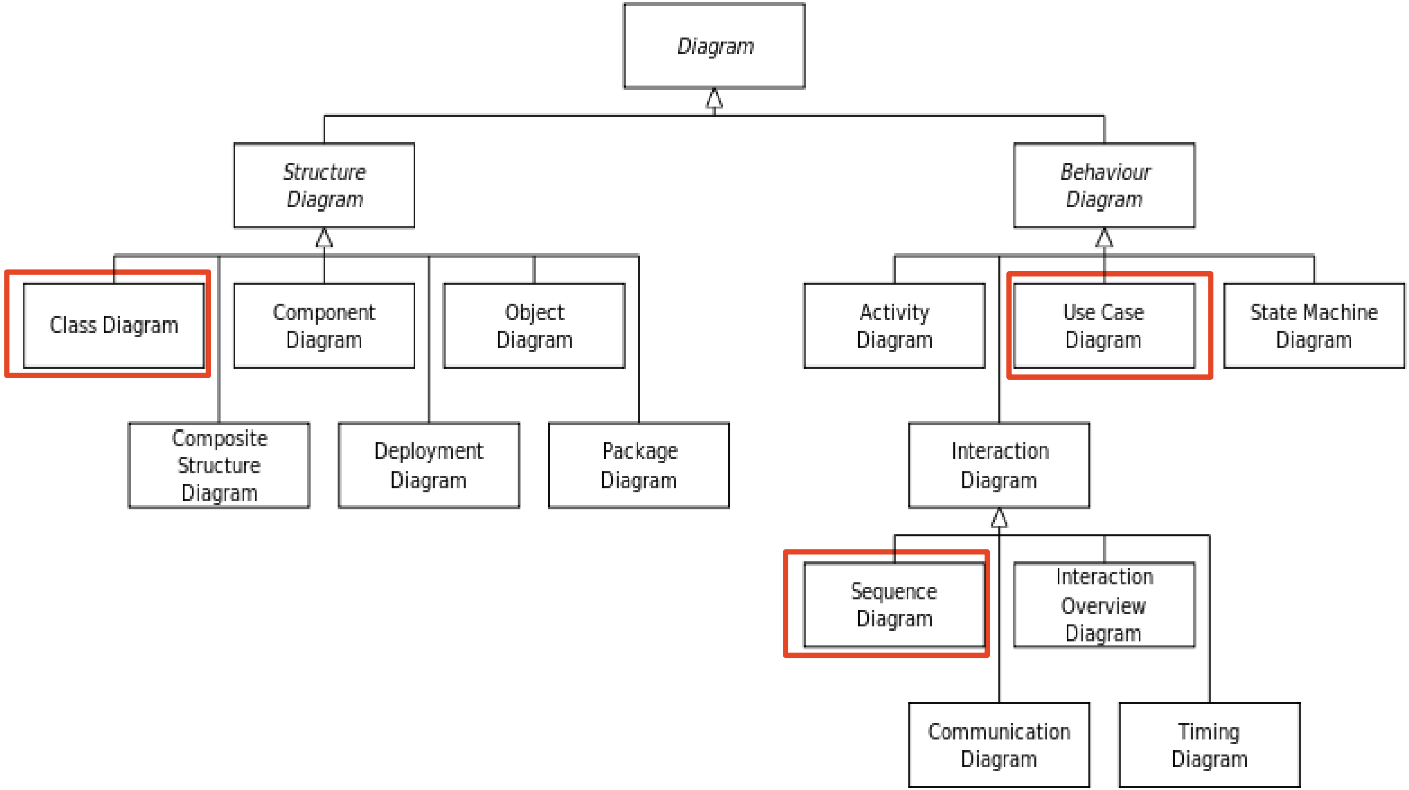
## Use Case Diagrams

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).*

THE UNIVERSITY OF SYDNEY

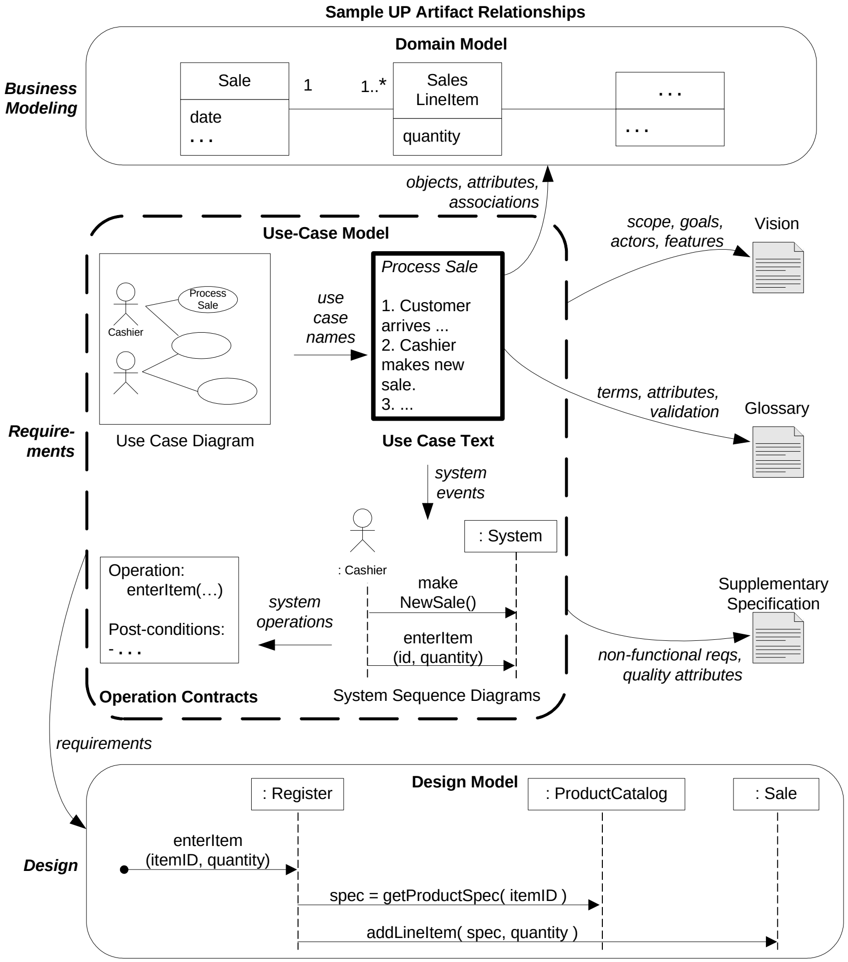


# Remember - UML Diagrams



https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/UML

# Business Modeling, Requirements and Designs in RUP/UP



# Use Cases

- **Use case:** "*specifies a set of behaviors performed by a system, which yields an observable result that is of value for Actors or other stakeholders of the system*"*
  - It capture what a system supposed to do (system's requirements)
  - Text documents not diagrams
  - Primary part of the use case model

- **Scenario (use case instance):** specific sequence of action and interactions between actors and the system
  - One particular story of using a system (e.g., successfully purchasing items with cash)

* OMG Unified Modeling Language, version 2.5.1, Dec. 2017 https://www.omg.org/spec/UML/2.5.1

# Use Case Diagrams

- UML graphical notations that help to capture uses cases (system’s boundaries and interactions and relationships)

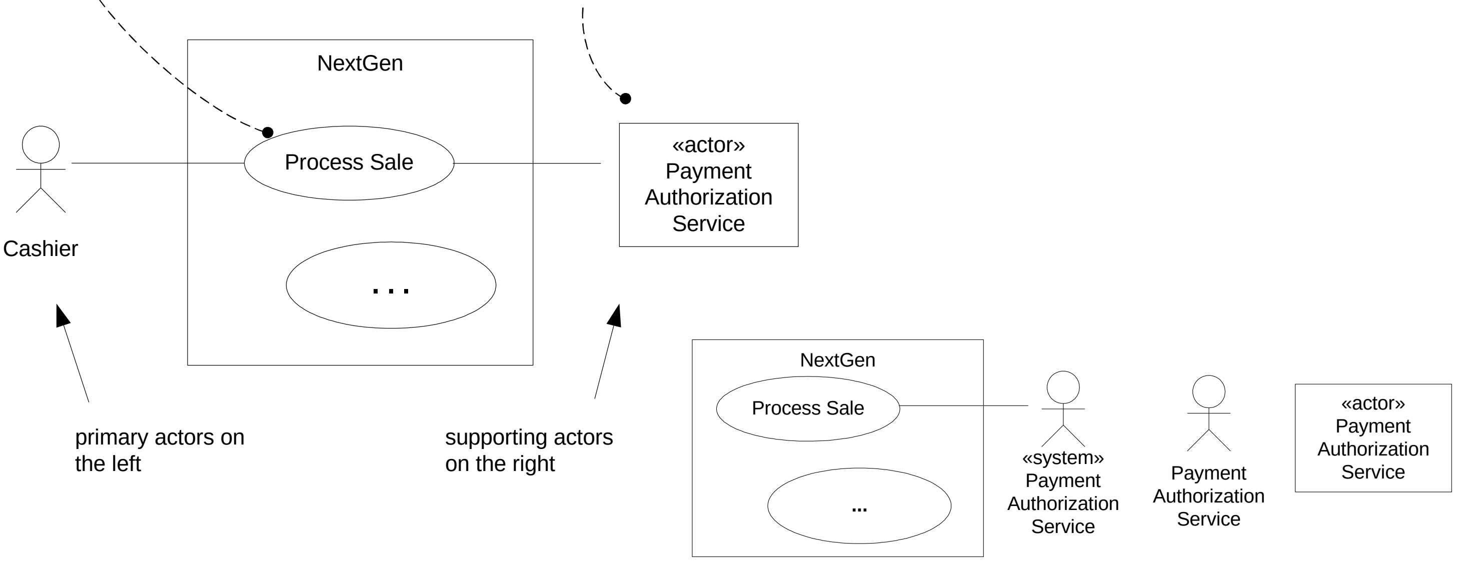
  - **Subject:** system under consideration to which the use case applies
  - **Actor:** role that interact with the subject/system (e.g., end user, customer, supplier, another system)
  - **Use case:** describes functionality of the system
  - Association: relationship between an actor and a use case (an actor can use certain functionality of the system)
    - «include» indicates the behavior of the included use case is included in the behavior of the including use case

* OMG Unified Modeling Language, version 2.5.1, Dec. 2017 https://www.omg.org/spec/UML/2.5.1

# Use Case Diagram – UML Notations

For a use case context diagram, limit the use cases to user-goal level use cases.

Show computer system actors with an alternate notation to human actors.



primary actors on the left

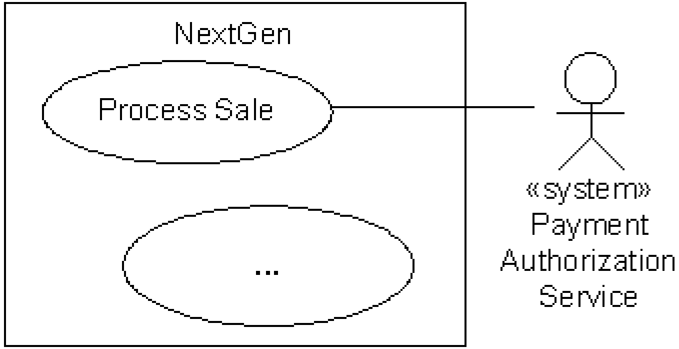
supporting actors on the right

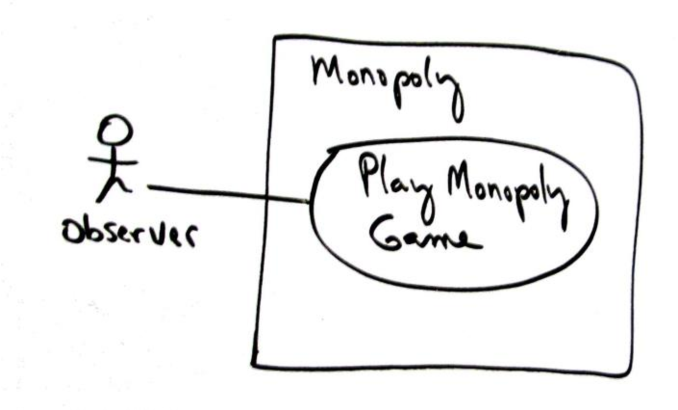
Some UML alternatives to illustrate external actors that are other computer systems.

The class box style can be used for any actor, computer or human. Using it for computer actors provides visual distinction.

# Use Case Diagrams – Tools

- There are many tools to aid drawing UML diagrams
  - Tools are means to make your life easier
  - You can also draw diagrams using pen-an-paper or white board



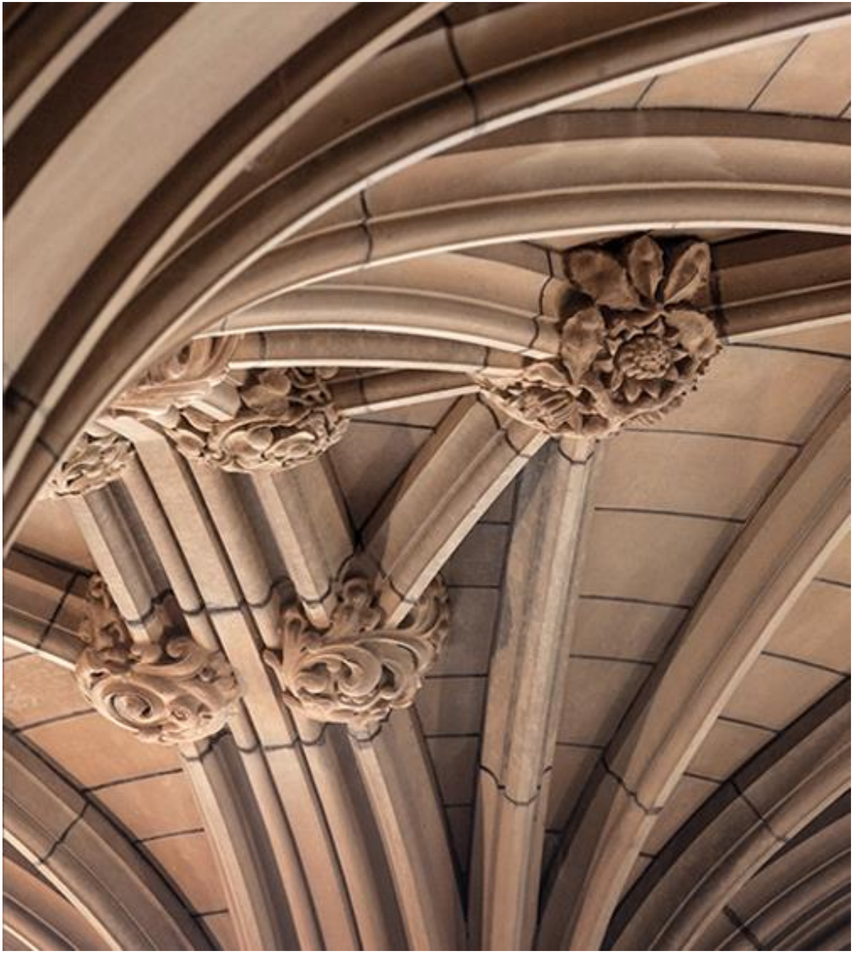


https://www.lucidchart.com/pages/examples/uml_diagram_tool

# UML Modeling – Class Diagrams

## Structural Diagrams

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*.

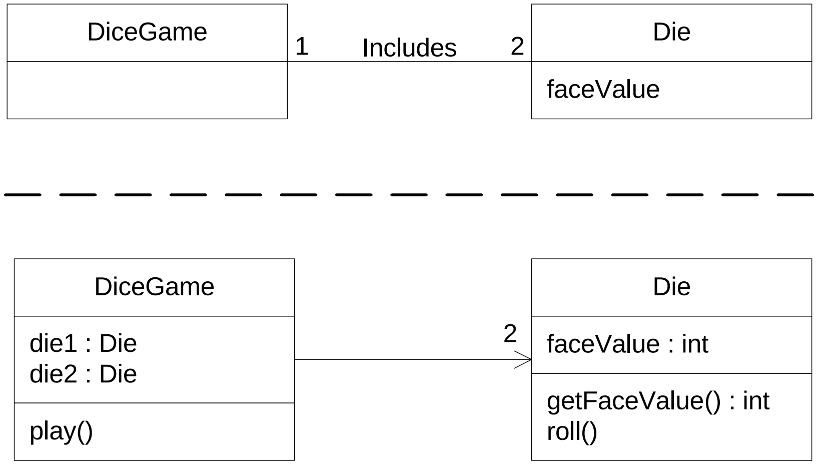


# Class Diagram – Perspectives

- **Conceptual:** describes key concepts in the problem domain. Use in business modeling for OO analysis

- **Specification:** describes software components with specification and interfaces

- **Implementation:** describes software implementation in a particular programming language (e.g., Java)



**Conceptual Perspective (domain model)**

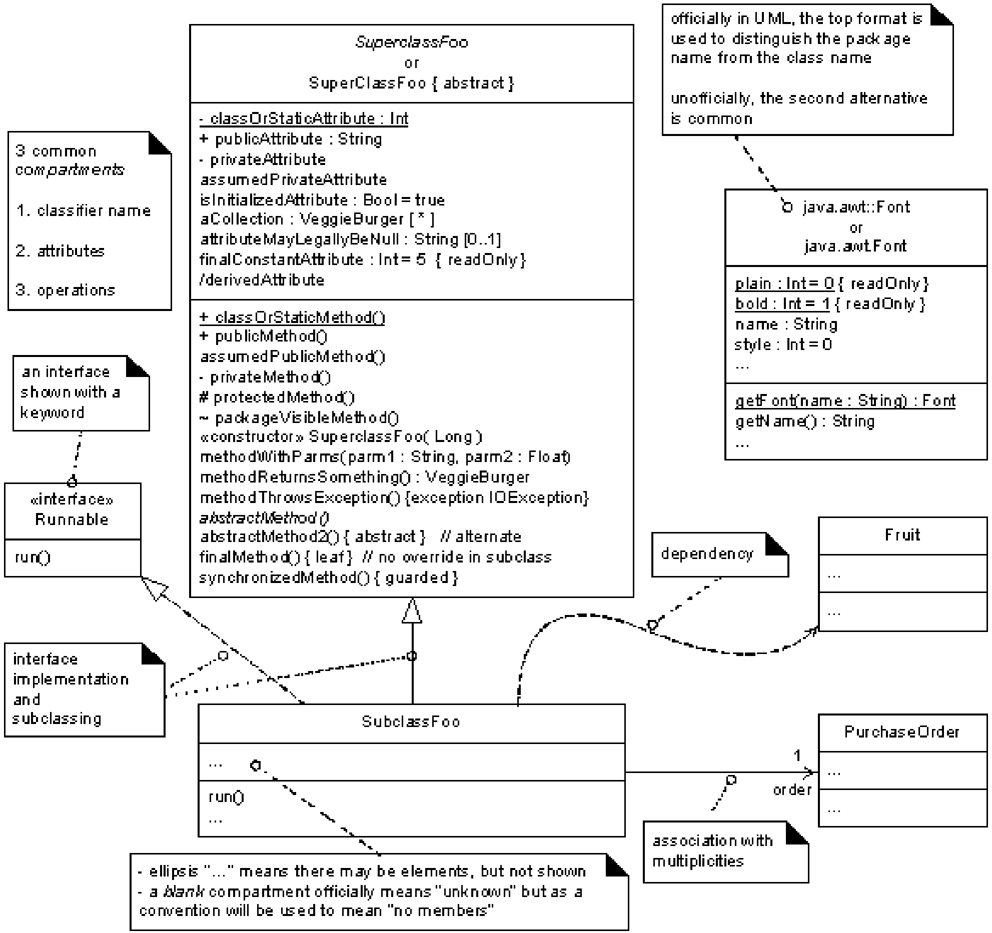
Raw UML class diagram notation used to visualize real-world concepts.

**Specification or Implementation Perspective (design class diagram)**

Raw UML class diagram notation used to visualize software elements.

# Class Diagram – Notations

- Common compartments; classifier name, attributes and operations
    - Package name
    - <<interface>>
- Dependency
- Class hierarchy – inheritance
- Association and multiplicity
- Optional and default elements



*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 }

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

unofficially, the second alternative is common

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
...

3 common *compartments*

1. classifier name

2. attributes

3. operations

an interface shown with a keyword

«interface»
Runnable

run()

interface implementation and subclassing

SubclassFoo

...

run()
...

dependency

Fruit

...

...

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"

# Class Diagrams – UML Attributes

**Attribute Text**

*Visibility : type : default {property string}*

Visibility + (public), - (private)

Attributes are assumed private if no visibility sign shown

**Attribute-as-association**

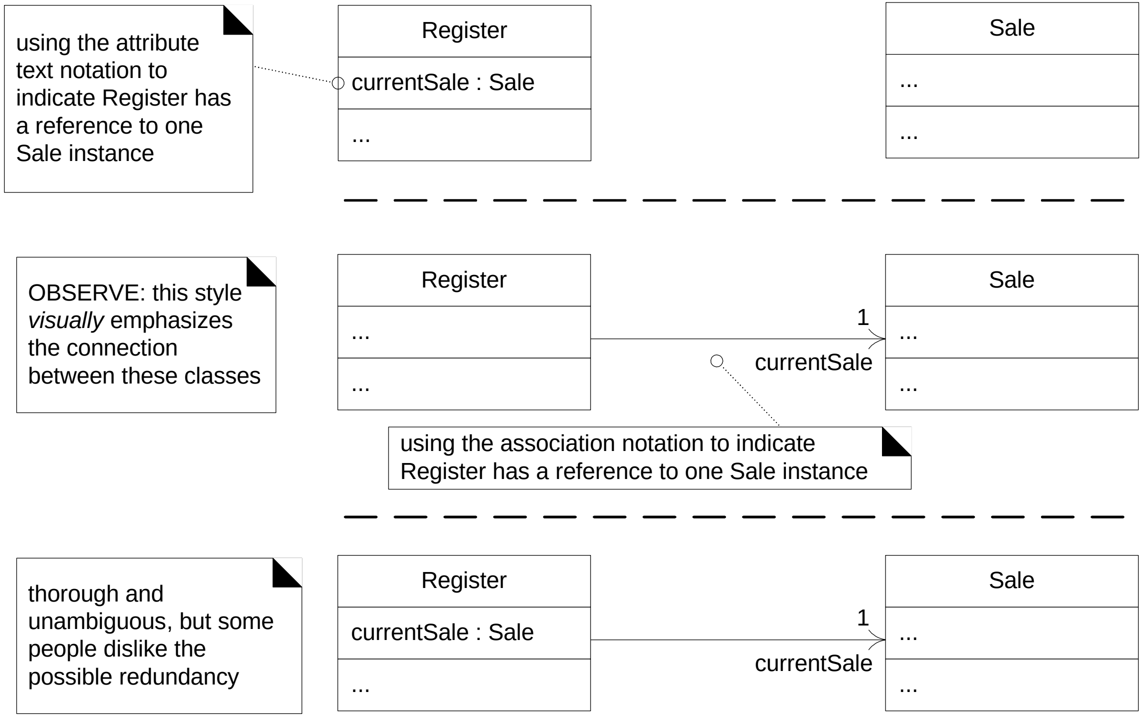
Arrow pointing from the source to the target

Multiplicity and '*rolename*' (*currentSale*) at the target

No association name

**Attribute text and as-association**

- Not popular



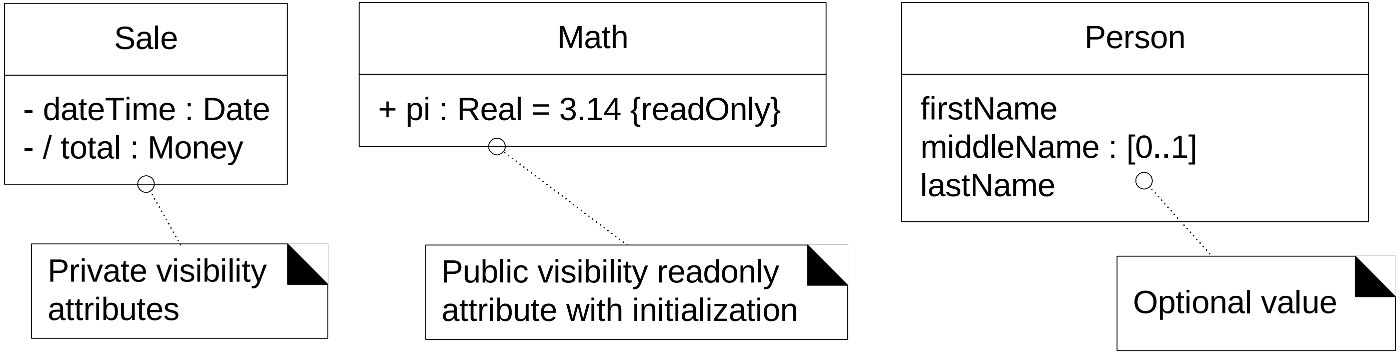
using the attribute text notation to indicate Register has a reference to one Sale instance

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

# Class Diagrams – Attributes


Sale
- dateTime : Date
- / total : Money
Private visibility attributes
Math
+ pi : Real = 3.14 {readOnly}
Public visibility readonly attribute with initialization
Person
firstName
middleName : [0..1]
lastName
Optional value

Read-only Attributes with initialization, and optional values

# Class Diagrams – Operations

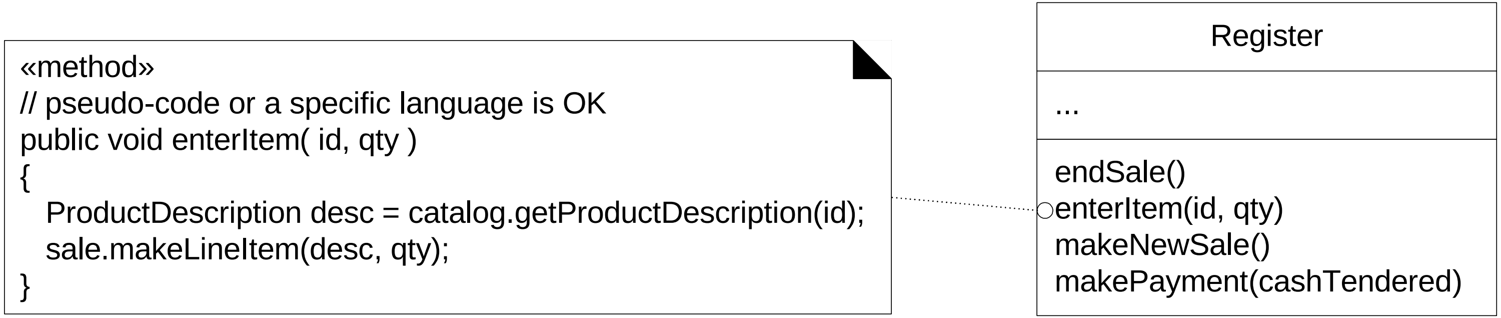
**Visibility (parameter-list) : return-type {property-string}** (UML 1)
**Visibility (parameter-list) {property-string}** (UML 2)

- Not a method, but declaration with a name, parameters, return type, exception list, and possibly a set of constraints of pre-and post-conditions
- Operations are public by default, if no visibility shown
- Operation signature in a programming language is allowed, e.g.,

*+ getPlayer (name : String) : Player {exception IOException}*
*Public Player getPlayer(String name) throws IOException*

# Class Diagrams – Methods

- Implementation of an operation, can be specified in:
  - Class diagrams using UML *note* symbol with stereotype symbol «*method*»
    - Mixing static view (class diagram) and dynamic view (method implementation)
    - Good for code generation (forward engineering)
  - Interaction diagrams by the details and sequence of messages



```
«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)

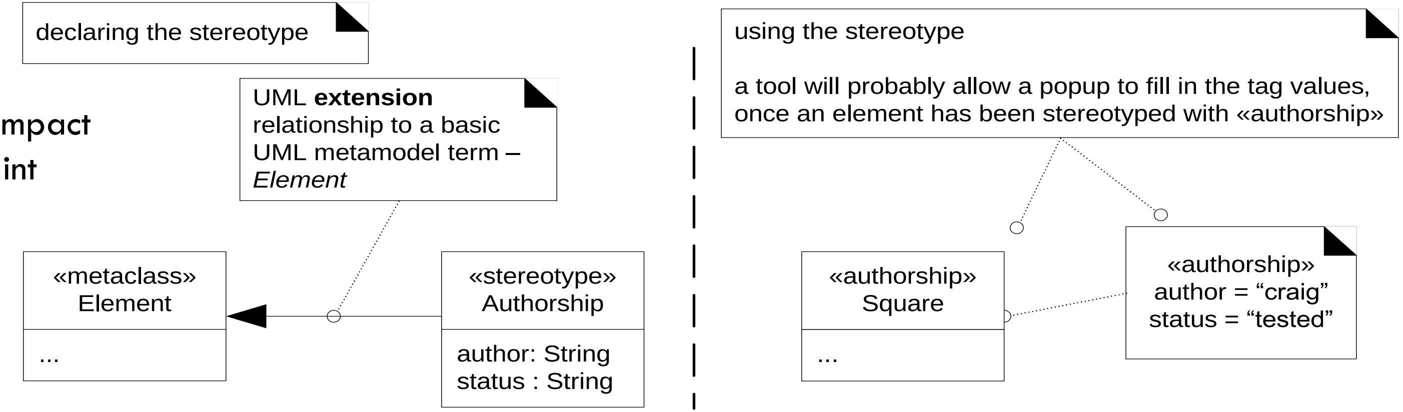
# UML Keywords

- Textual adornment to categorize a model element
  - Using «» or { }
  - UML 2 the brackets («») are used for keywords and stereotype

| Keyword | Meaning | Example usage |
|---|---|---|
| «*interface*» | Classifier is an interface | In class diagram, above classifier name |
| {abstract} | Abstract element; can't be instantiated | In class diagrams, after classifier name or operation name |
| {ordered} | A set of objects have some imposed ordered | In class diagrams, at an association end |

# UML Stereotypes

- **Stereotypes** allow refinement (extension) of an existing modeling concept
  - Defined in UML Profile
- **UML profile:** group of related model elements allow customizing UML models for a specific domain or platform
  - Extends UML's «metaclass» *Element*
- **UML note symbol**
  - Has no semantic impact
    Specify a constraint



declaring the stereotype
UML **extension** relationship to a basic UML metamodel term – *Element*
«metaclass» Element
...
«stereotype» Authorship
author: String
status : String
using the stereotype
a tool will probably allow a popup to fill in the tag values, once an element has been stereotyped with «authorship»
«authorship» Square
...
«authorship»
author = "craig"
status = "tested"

# Generalization, Abstract Classes & Operations

*"Generalization – a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, specific classifiers indirectly has features of the more general classifiers." [OMG2003]*

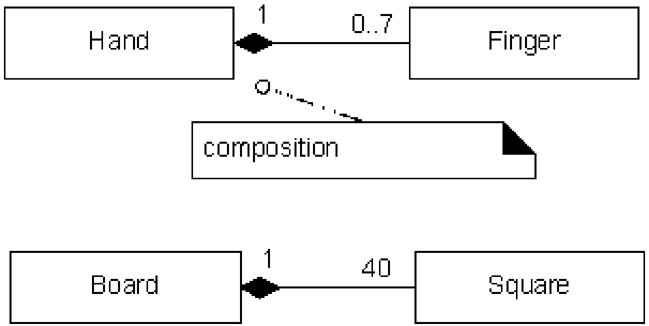
- Generalization implies inheritance in design class diagram (software perspective) but not in the domain model (conceptual perspective)

- Classes and operations with **{*abstract*}** tag are abstract

- Classes and operations with **{*leaf*}** *are* ***final*** (cannot be overridden in the sub-classes)

# Composition

- Composition, or composite aggregation, relationship implies:
  - Instance of the part (e.g., Square) belongs to only one composite instance at a time (e.g., one board)
  - The part must always belong to a composite
  - The composite is responsible for the creation and deletion of its parts (by itself or by collaborating with other objects)



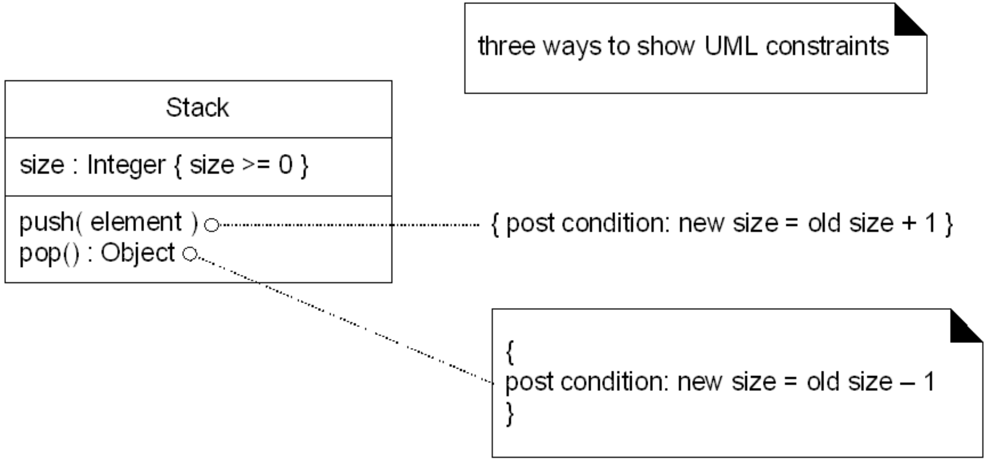
composition means
-a part instance (*Square*) can only be part of one composite (*Board*) at a time

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

# Constraints

- Restriction/condition on a UML elements described in a natural or a formal language (Object Constraint Language (OCL))

- Different ways to represent constraints



three ways to show UML constraints

Stack

size : Integer { size >= 0 }

push( element )
pop() : Object

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

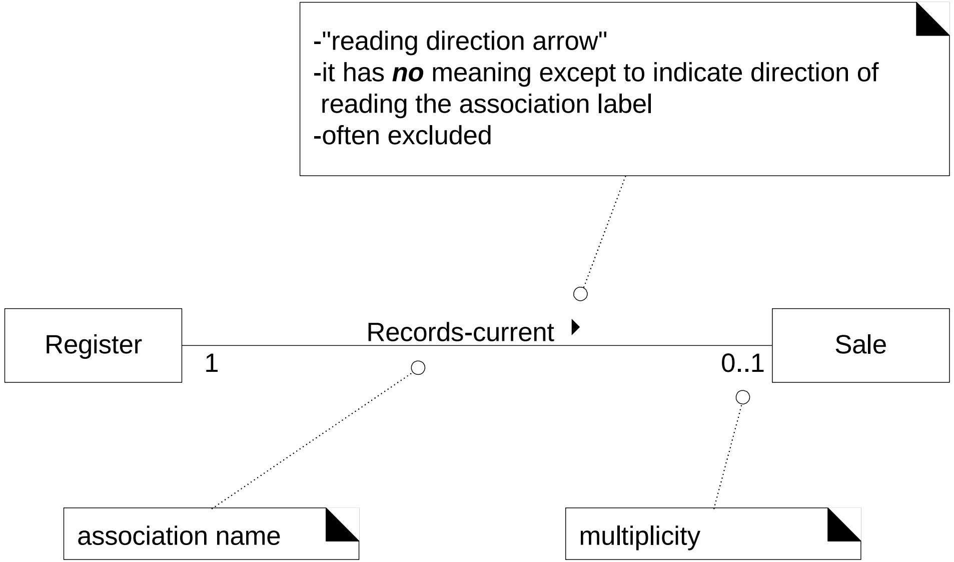
{
post condition: new size = old size – 1
}

# Associations

- Relationship between classifiers where logical or physical link exists among classifier’s instances

- May implemented differently; no certain construct linked with association

- Notations:
  - Association name (meaningful)
  - Multiplicity
  - Direction arrow



-"reading direction arrow"
-it has ***no*** meaning except to indicate direction of reading the association label
-often excluded
Register
Records-current ▸
Sale
1
0..1
association name
multiplicity

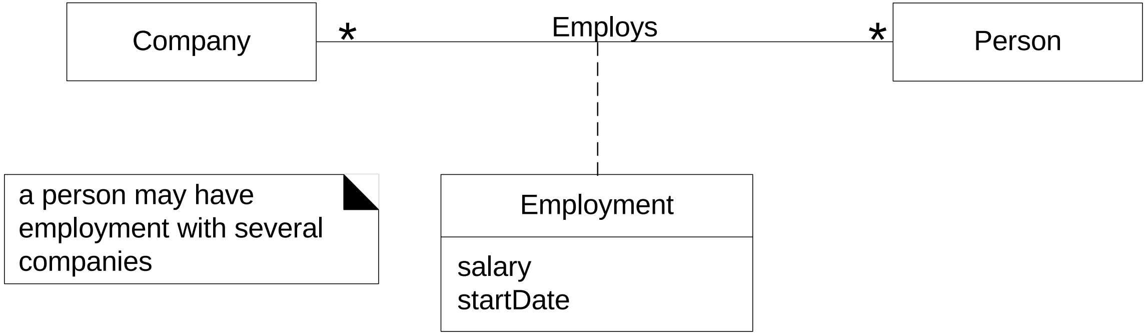
# Associations – Multiplicity

- Multiplicity; number of instances involved in the relationship (association)
- Communicates domain constraints that will be implemented
- Multiplicity focus on the relationship at a <u>particular moment</u>, rather than over a span of time
  - "In countries with monogamy laws, a person can be *Married-to* only <u>**one**</u> other person at any particular moment, even though over a span of time, that same person may be married to <u>**many**</u> persons."

| Multiplicity | Meaning (number of participating instances) |
|---|---|
| * | Zero or more; many |
| 0..1 | Zero or one |
| 1..* | One or more |
| 1..n | One to n |
| n | Exactly n |
| n, m, k | Exactly n, m or k |

# Association Class

- Modeling an association as a class (with attributes, operations & other features)
  - A Company *Employs* many Persons
  - *Employs* → *Employment* class with attributes salary and startDate



# Dependency

- A dependency exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client)
- Various reason for dependency
  - Class send message to another
  - One class has another as its data
  - One class mention another as a parameter to an operation
  - One class is a superclass or interface of another

# When to show dependency?

- Be selective in describing dependency
- Many dependencies are already shown in other format
- To depict global, parameter variable, local variable and static-method
- To show how changes in one element might alter other elements

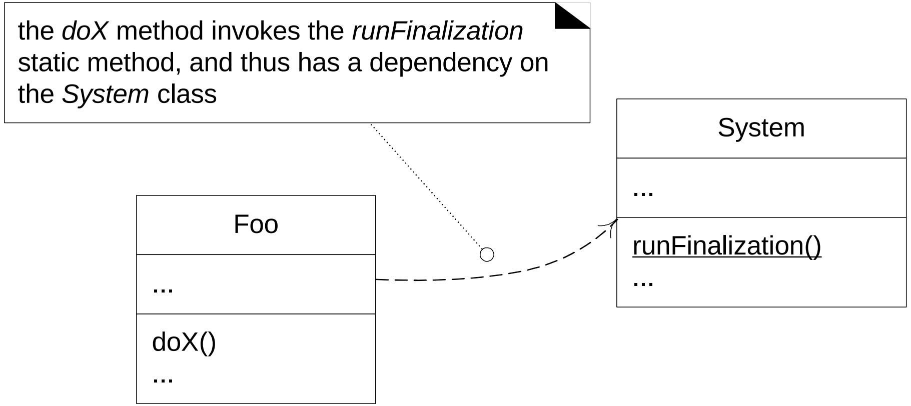
- There are many varieties of dependency, use keywords to differentiate them
- Different tools have different sets of supported dependency keywords:

<<call>> the source calls an operation in the target

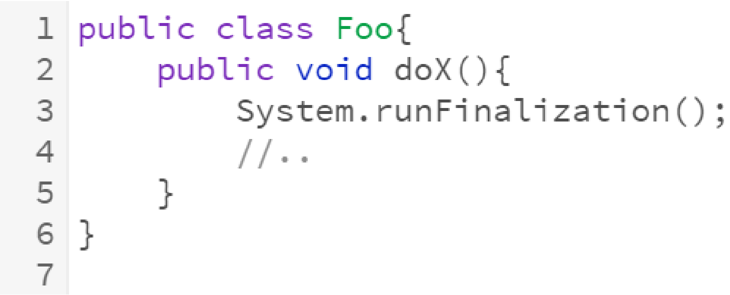
<<use>> the source requires the targets for its implementation

<<parameter>> the target is passed to the source as parameter.

# Dependency Example



the *doX* method invokes the *runFinalization* static method, and thus has a dependency on the *System* class

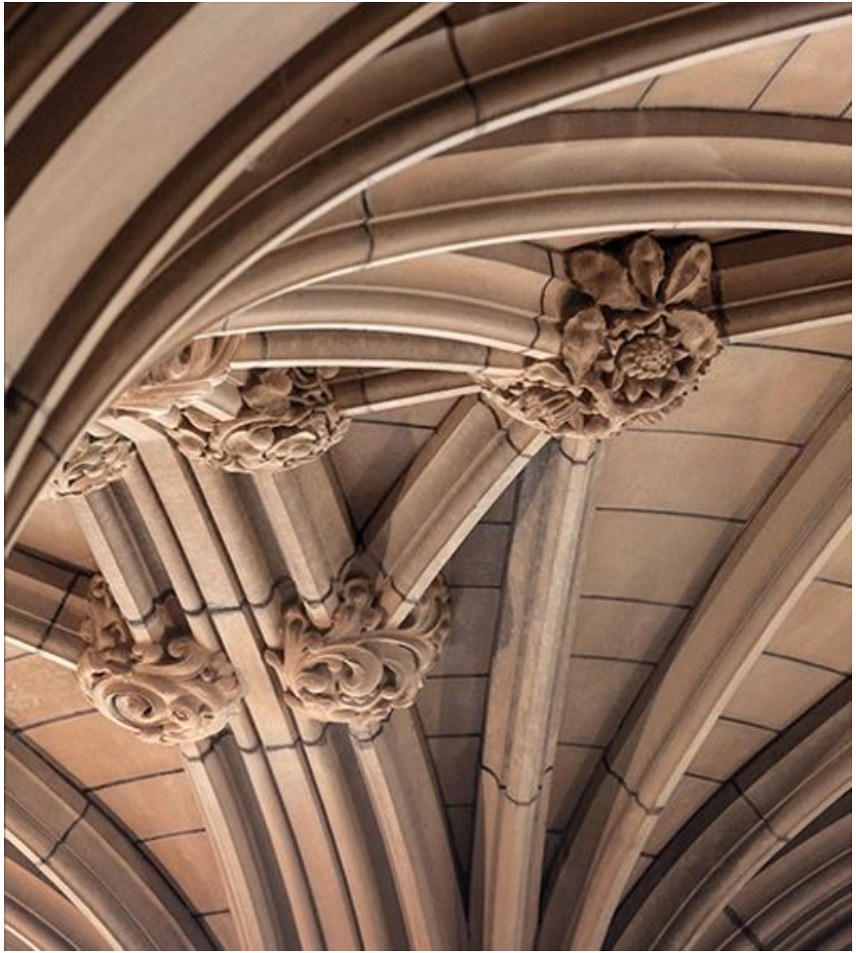


```
public class Foo{
    public void doX(){
        System.runFinalization();
        //..
    }
}

```

# UML Interaction Diagrams

## Dynamic (Behavioural) Diagrams

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*.

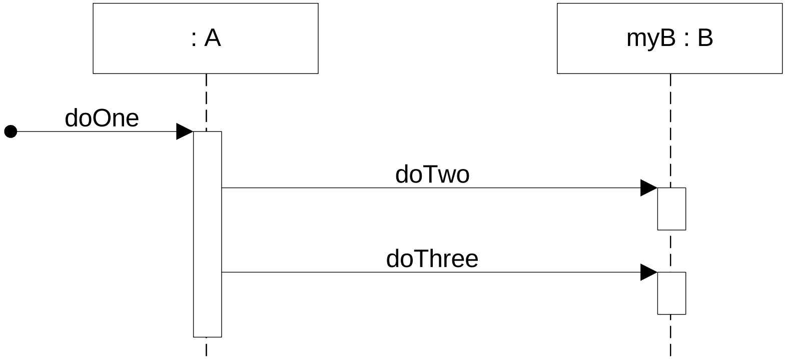


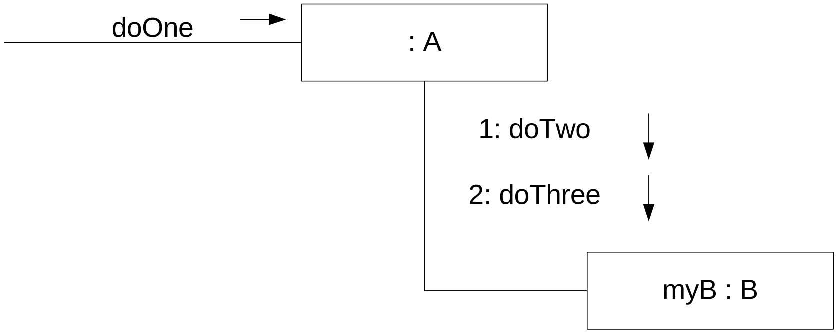
# UML Interaction Diagrams

- One of the dynamic (behavioral) diagrams which consists of diagrams including *Sequence* and *Communication* diagram

**Sequence diagrams:** illustrate sequence/time-ordering of messages in a fence format (each object is added to the right)

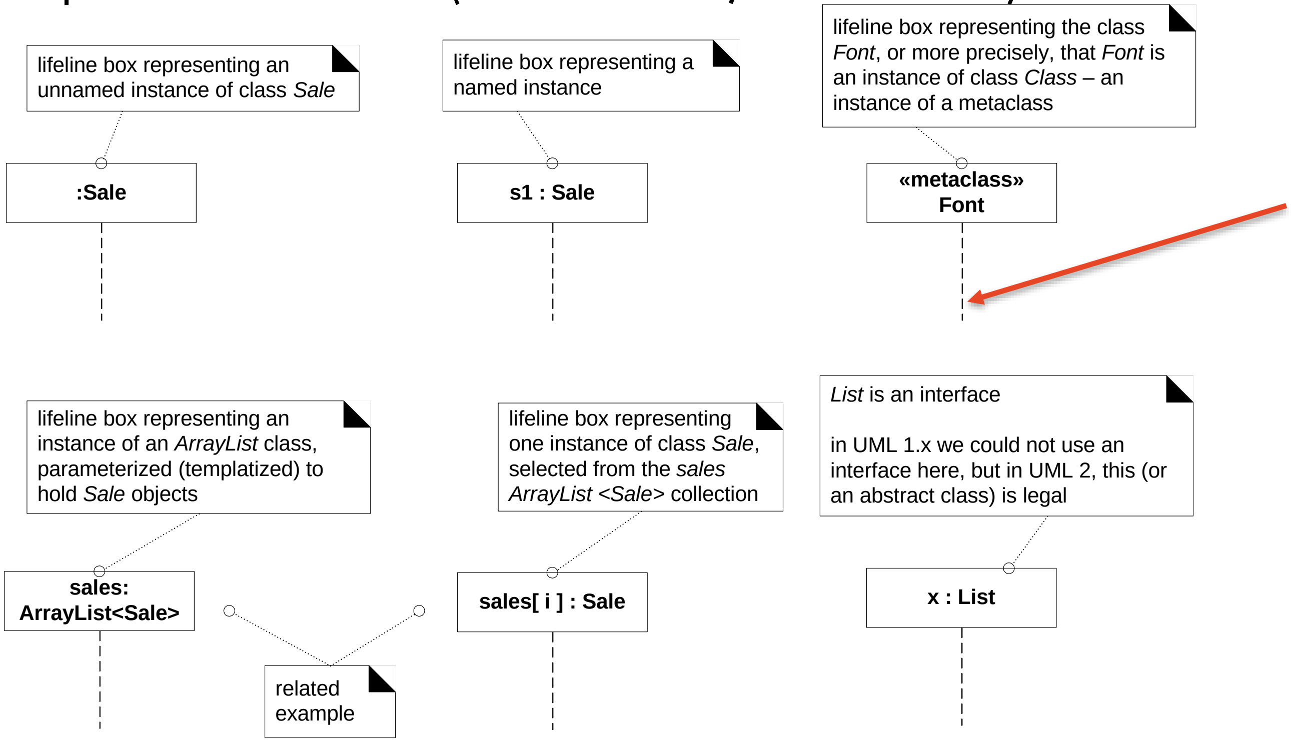
**Communication diagrams:** objects' interactions are illustrated in a graph/network format





# Sequence Diagrams: Classes/Objects

- Participants in interactions (class instances, lifeline boxes)



lifeline box representing an unnamed instance of class *Sale*

:Sale

lifeline box representing a named instance

s1 : Sale

lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass

«metaclass»
Font

lifeline box representing an instance of an *ArrayList* class, parameterized (templatized) to hold *Sale* objects

sales:
ArrayList<Sale>

related example

lifeline box representing one instance of class *Sale*, selected from the *sales ArrayList <Sale>* collection

sales[ i ] : Sale

*List* is an interface

in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal

x : List

Actual lifeline of a participating object (solid or dashed)

# Sequence Diagrams: Messages

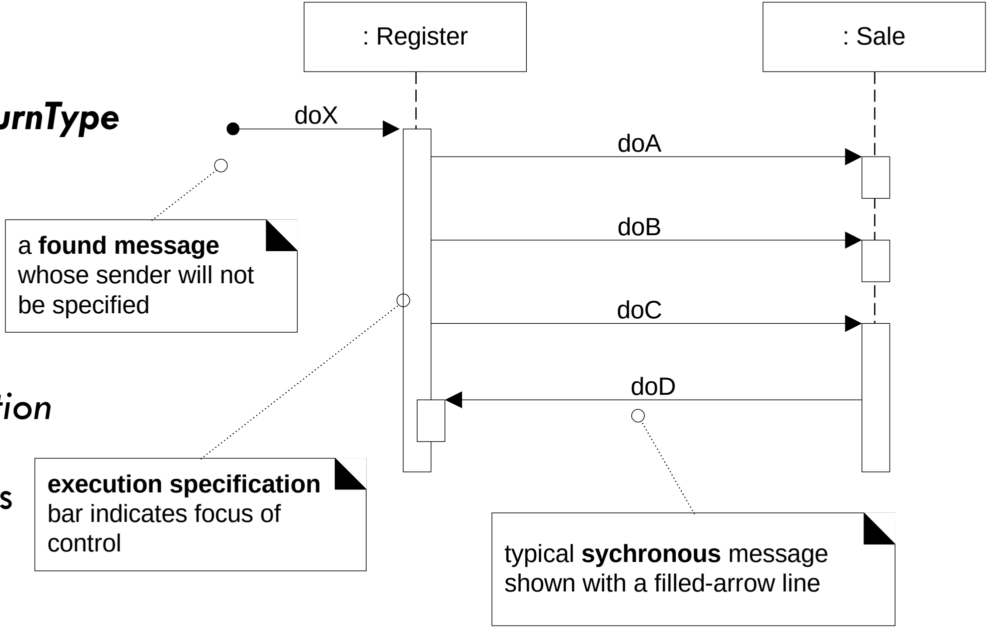
- Standard message syntax in UML

***Return = message (parameter : parameterType) : returnType***

- Some details may be excluded. **Examples:**

*Initialize(code)*
*descrp = getProductDescription(id)*
*descrp = getProductDescription(id) : ProductDescription*

- The time ordering from top to bottom of lifelines

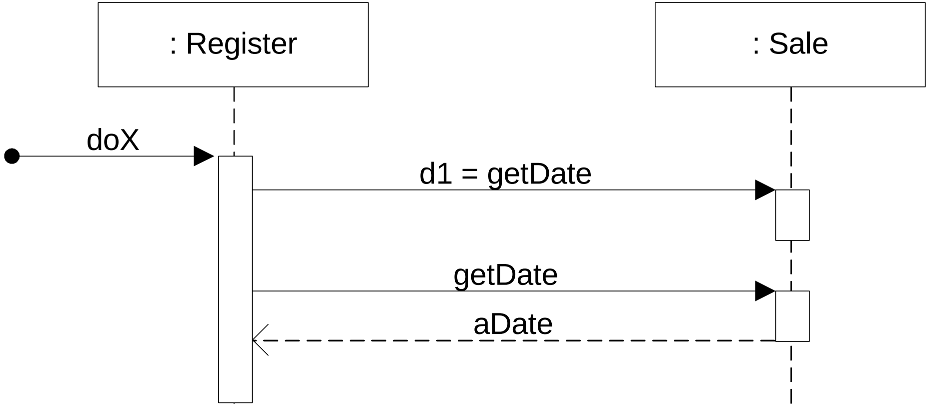


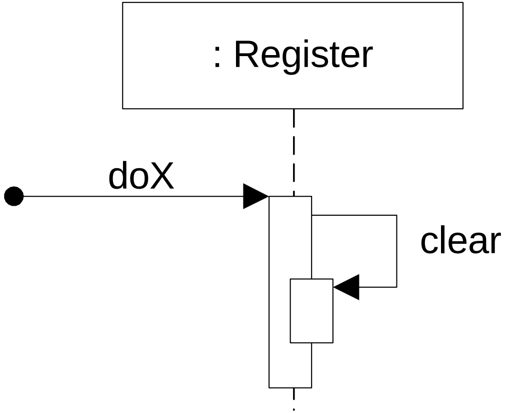
a **found message** whose sender will not be specified

**execution specification** bar indicates focus of control

typical **sychronous** message shown with a filled-arrow line

# Sequence Diagrams: Messages





**Message reply/return**

**1.** Standard reply/return message syntax

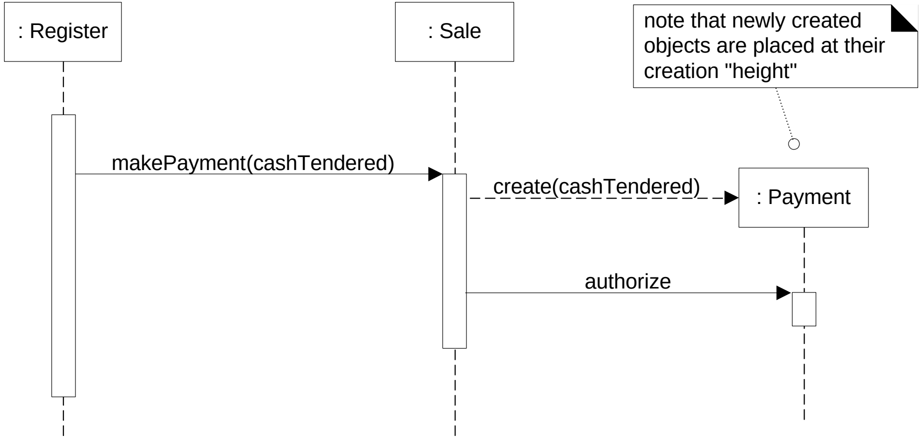
***returnVar = message (parameter)***

**2.** Reply/return message line at the end of execution bar

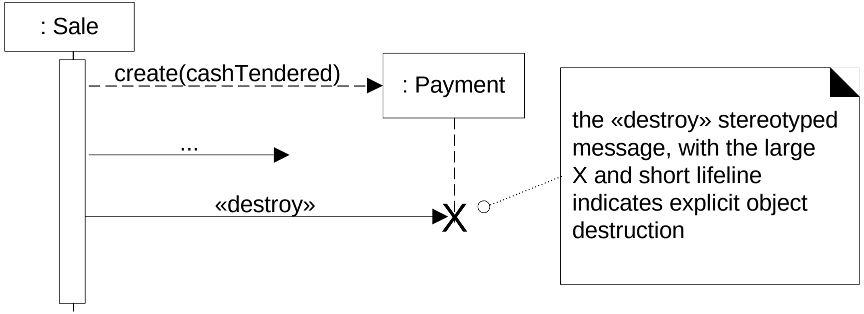
**Messages to "Self"**

Using nested execution bar

# Sequence Diagrams: Objects Creation/Destruction



note that newly created objects are placed at their creation "height"



the «destroy» stereotyped message, with the large X and short lifeline indicates explicit object destruction

**Object Creation**

Read as:

*"Invoke the new operator and call the constructor"*

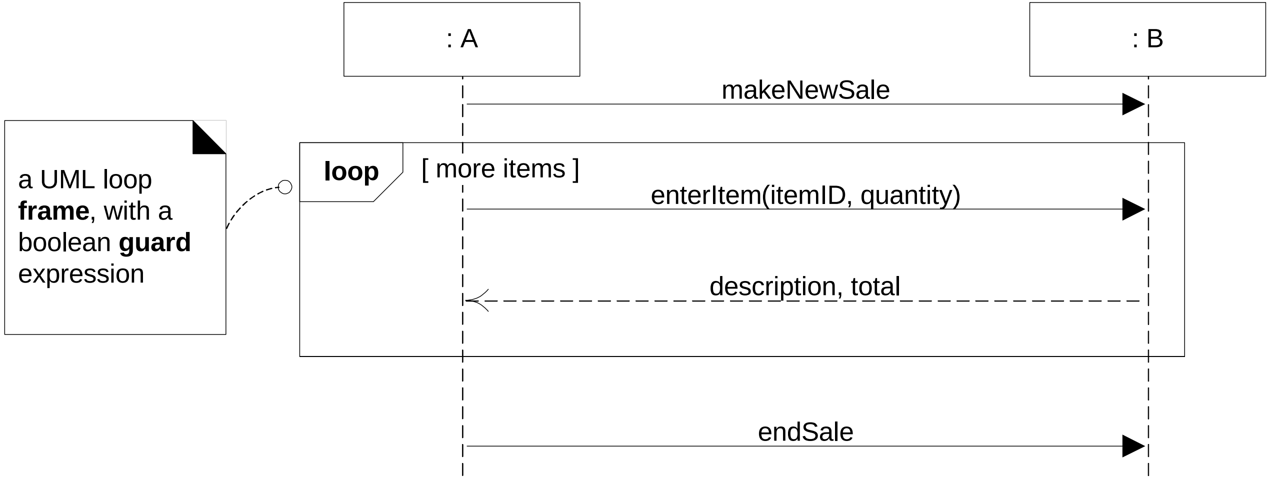
Message name is optional

**Object Destruction**

Explicit destruction to indicate object is no longer useable (e.g., closed database connection)

# Sequence Diagrams: Frames

- Diagram frames in UML sequence diagrams
  - Support conditional and looping construct



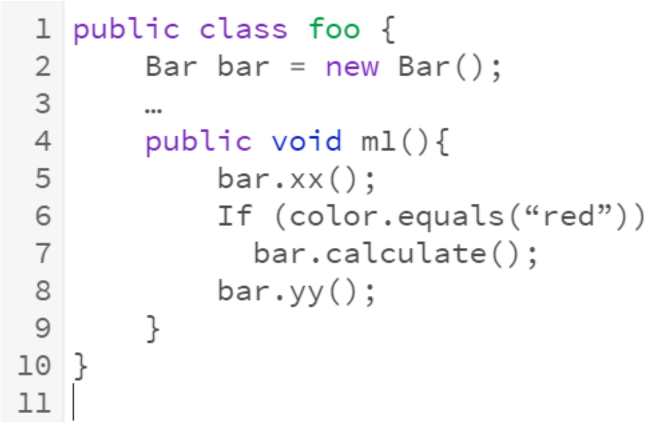
a UML loop **frame**, with a boolean **guard** expression

# Sequence Diagrams: Frames

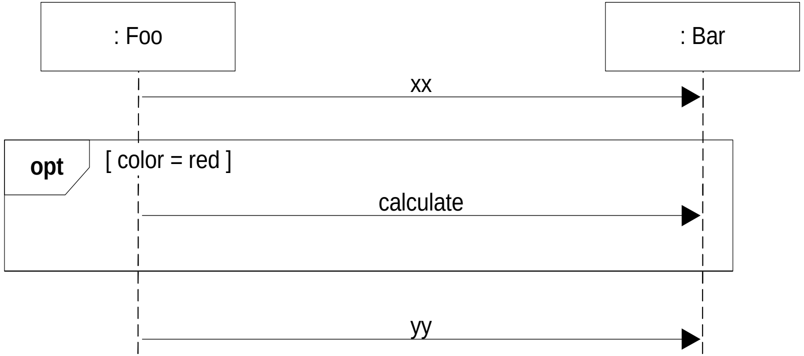
- Common frame operators

| **Frame operator** | **Meaning** |
|---|---|
| Alt | Alternative fragment for mutual exclusion conditional logic expressed in the guards |
| Loop | Loop fragment while guard is true |
| Opt | Optional fragment that executes if guard is true |
| Par | Parallel fragments that execute in parallel |
| Region | Critical region within which only one thread can run |

# Sequence Diagrams: Conditional Messages

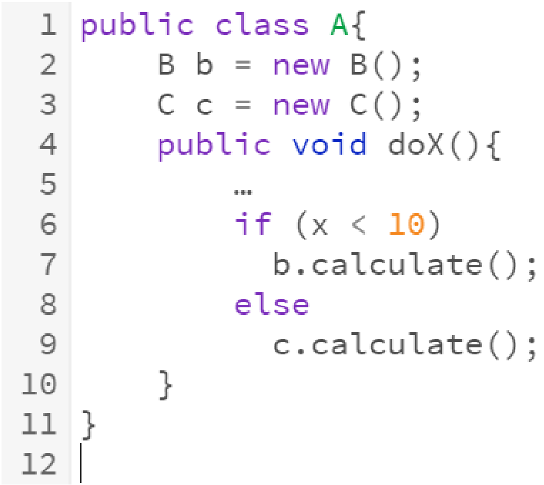


```
public class foo {
    Bar bar = new Bar();
    …
    public void m1(){
        bar.xx();
        If (color.equals(“red”))
          bar.calculate();
        bar.yy();
    }
}

```

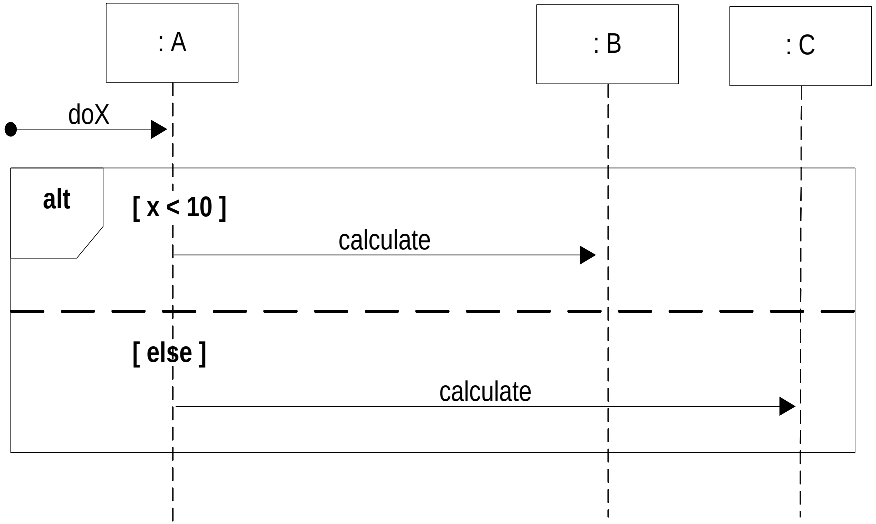


# Sequence Diagrams: Conditional Messages

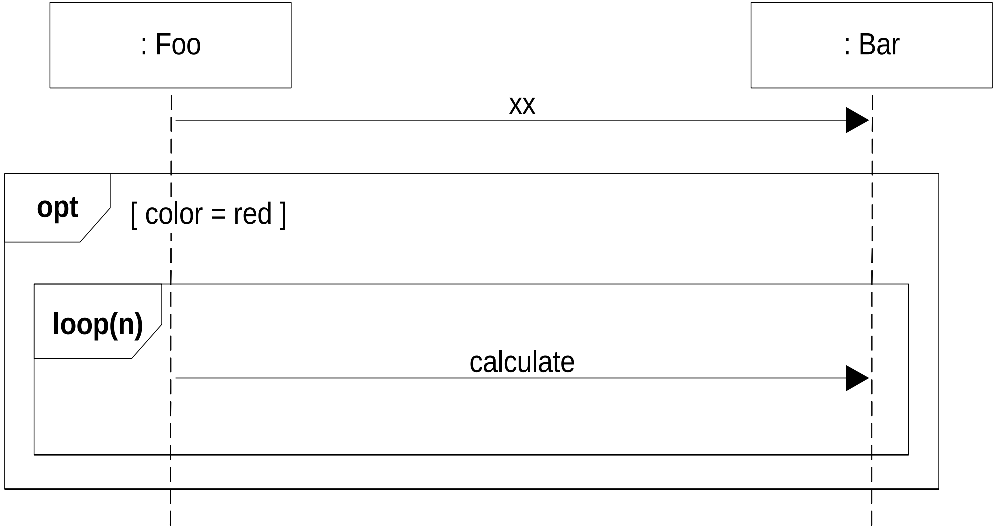
Mutually exclusive conditional messages



```
public class A{
     B b = new B();
     C c = new C();
     public void doX(){
          …
          if (x < 10)
            b.calculate();
          else
            c.calculate();
     }
}

```



# Sequence Diagrams: Nesting of Frames



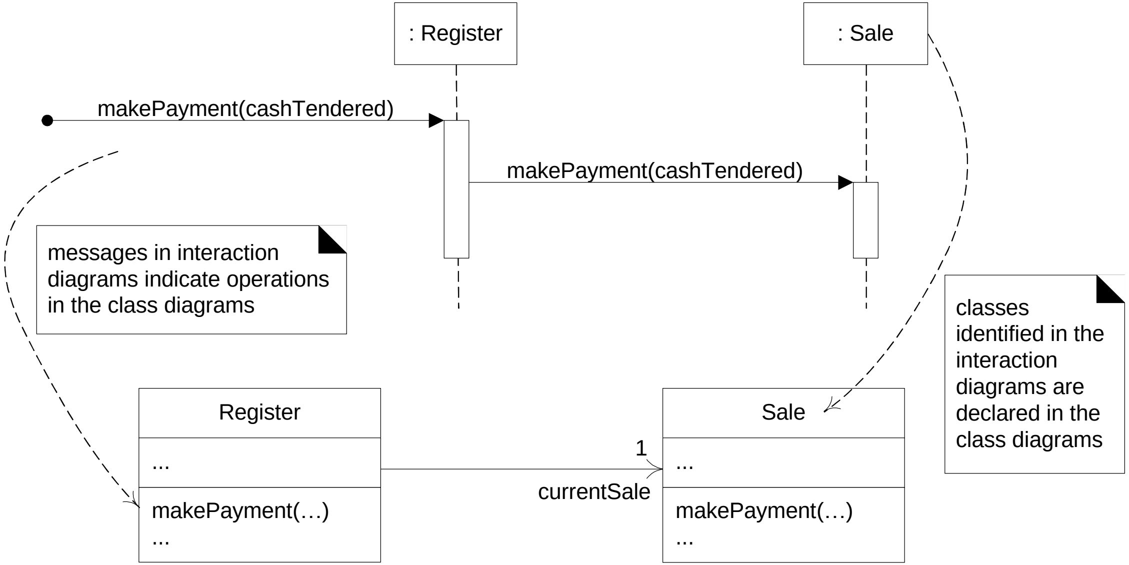
# Software Construction / Implementation

- Realization of design to produce a working software system
  - Meet customer requirements
- Design and implementation activities often interleaved
  - Agile development to accommodate for changes
- Object-Oriented design and Implementation model
  - Encapsulation
  - Abstraction
  - Reuse
  - Maintenance

# Class Diagrams: Relationship to Interaction Diagrams

- Interaction diagrams illustrates how objects interact via messages (dynamic behavior)
  - Classes and their methods can be derived
  - E.g., **Register** and **Sale** classes from *makePayment* sequence diagram

- Agile modeling practice: draw diagrams concurrently as dynamic and static views complement each other during the design process



messages in interaction diagrams indicate operations in the class diagrams

classes identified in the interaction diagrams are declared in the class diagrams

# Software Modelling Case Study

NextGen POS software modeling

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).*

# Next Gen Point-of-Sale (POS) System

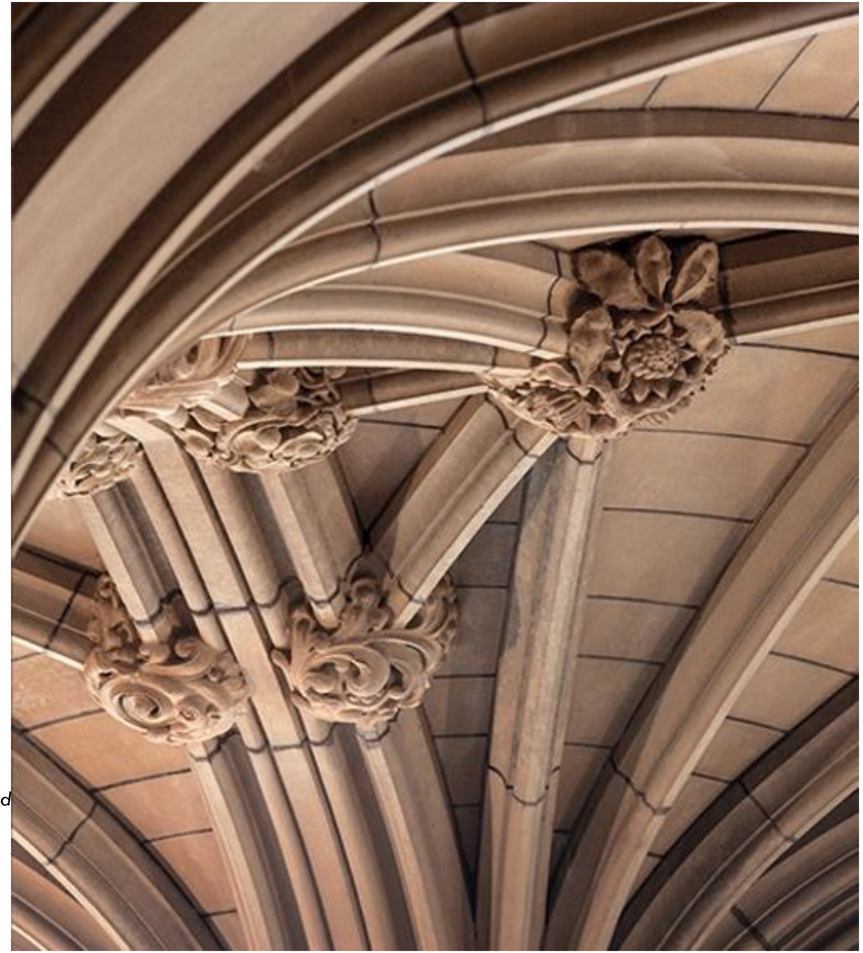
- A POS is a computerized application used (in part) to record sales and handle payments
  - Hardware: computer, bar code scanner
  - Software
  - Interfaces to service applications: tax calculator, inventory control
  - Must be fault-tolerant (can capture sales and handle cash payments even if remote services are temporarily unavailable
  - Must support multiple client-side terminals and interfaces; web browser terminal, PC with appropriate GUI, touch screen input, and Wireless PDAs
  - Used by small businesses in different scenarios such as initiation of new sales, adding new line item, etc.

# Next Gen POS Analysis

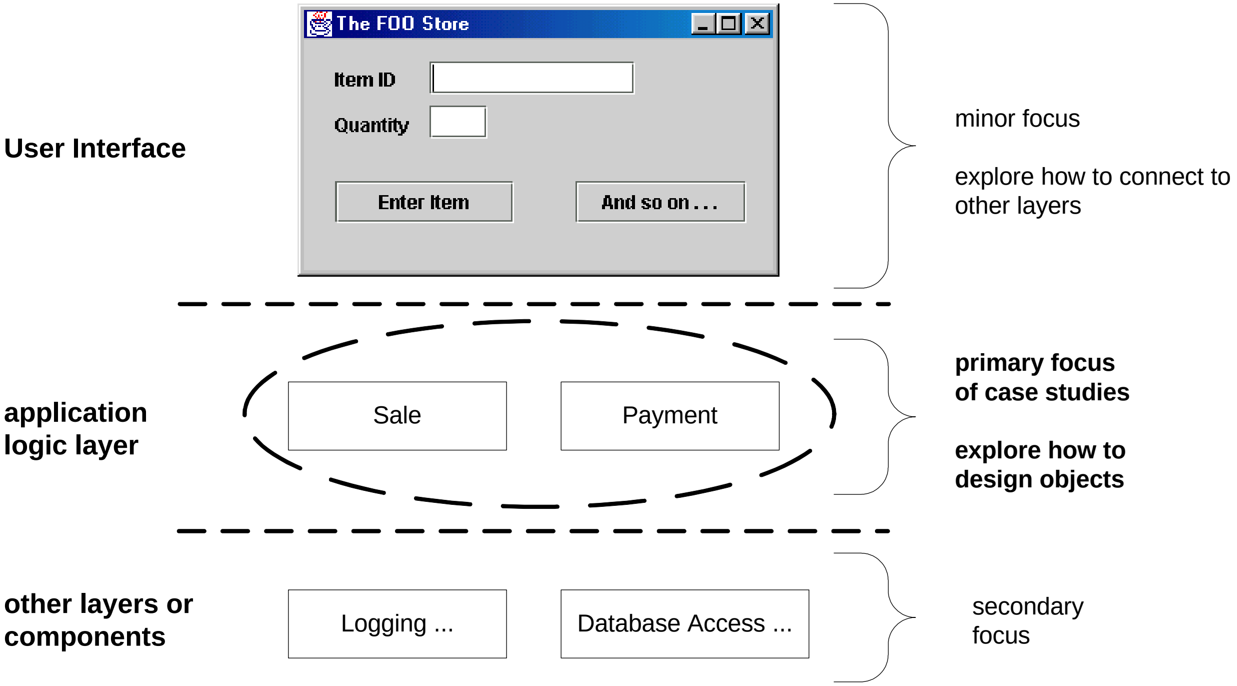
## Scope of OOA & D and Process Iteration

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).*

THE UNIVERSITY OF SYDNEY



# Next Gen POS – Scope (Analysis & Design)



**User Interface**

minor focus

explore how to connect to other layers

**application logic layer**

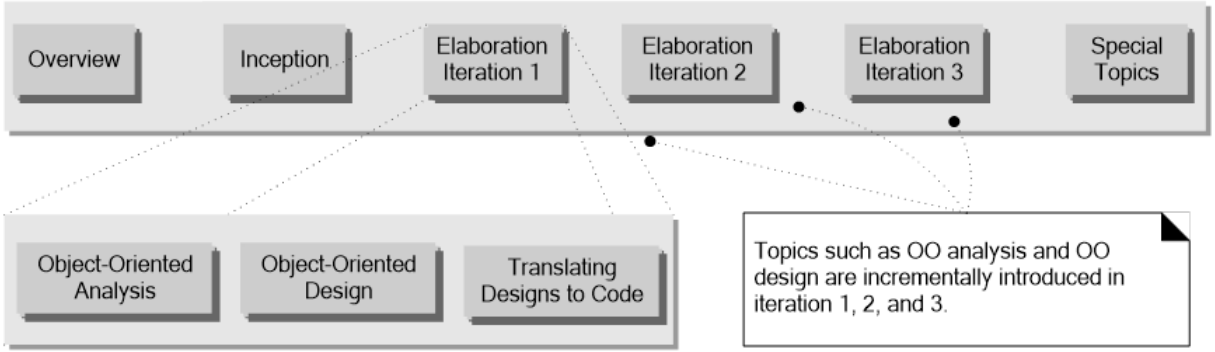
**primary focus of case studies**

**explore how to design objects**

**other layers or components**

secondary focus

# Iteration and Scope – Design and Construction



Topics such as OO analysis and OO design are incrementally introduced in iteration 1, 2, and 3.

Iteration 1

Introduces just those analysis and design skills related to iteration one.

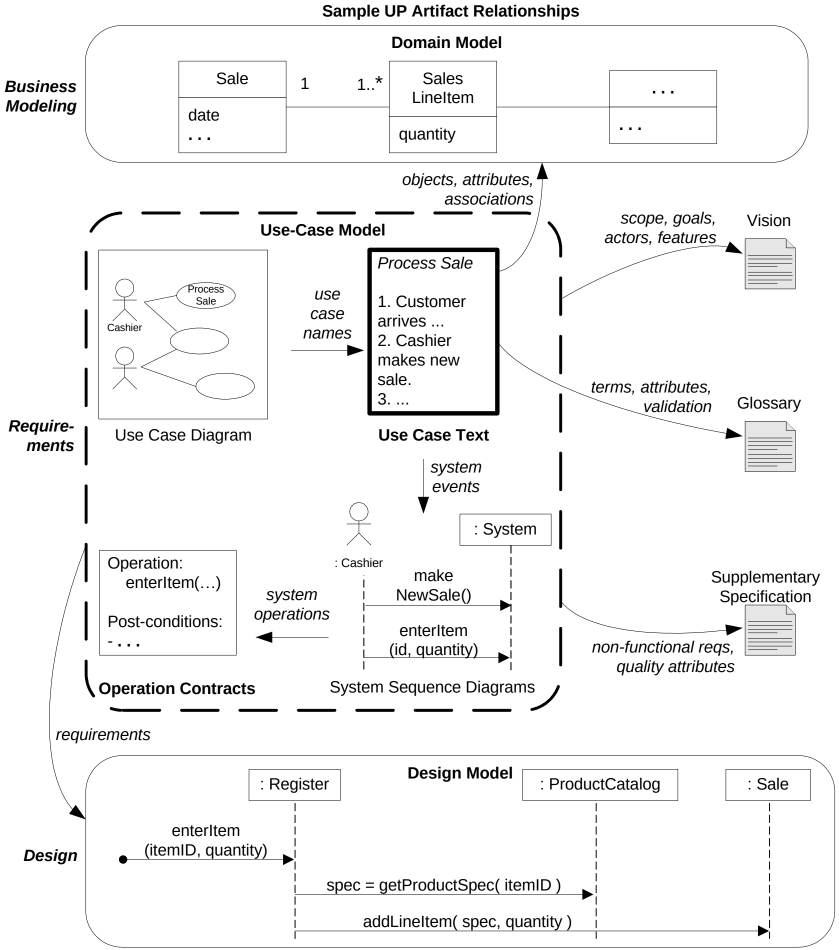
Iteration 2

Additional analysis and design skills introduced.

Iteration 3

Likewise.

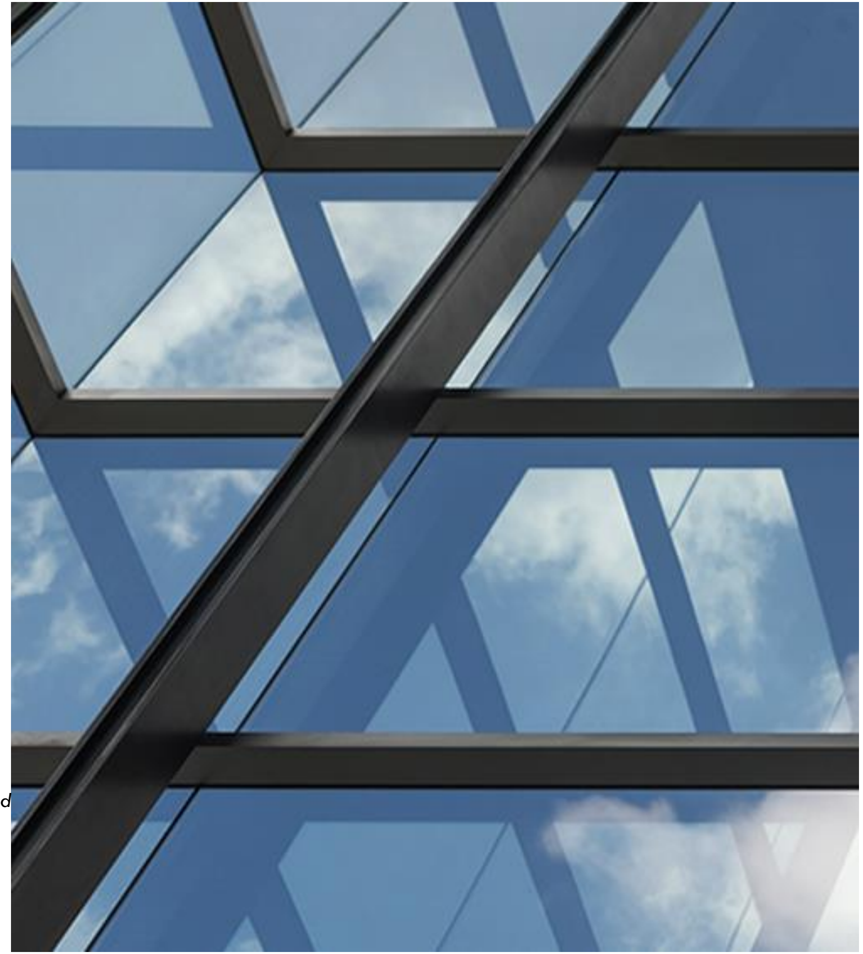
# Main Activities and Artefacts – Iterative Process



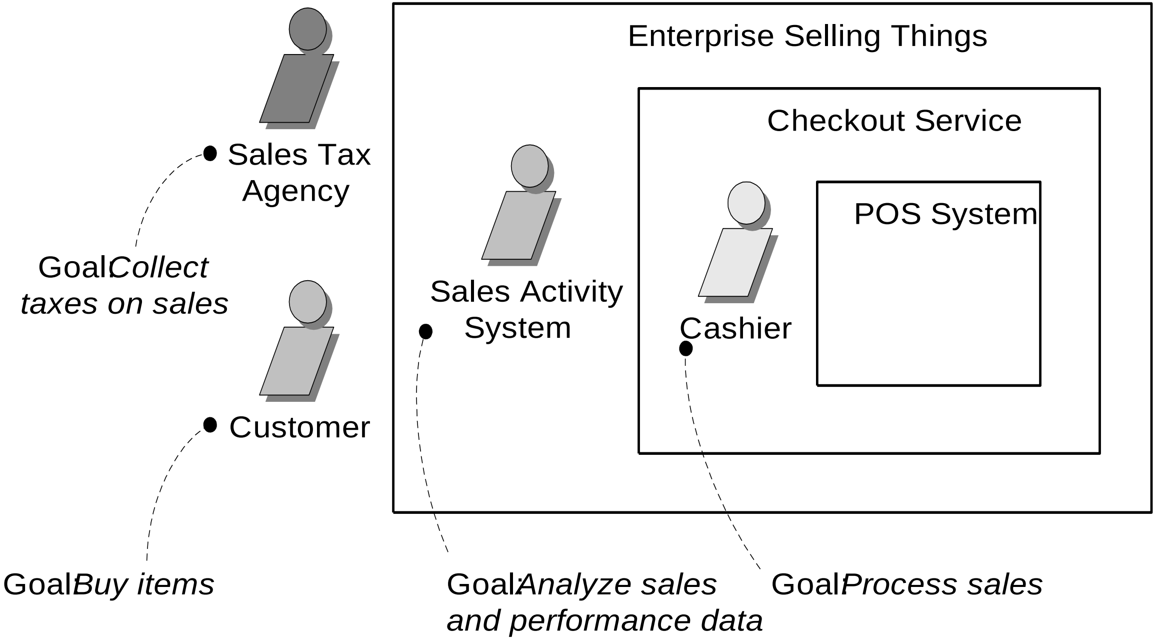
# Next Gen POS Case Study: Analysis

## OO Analysis with UML

Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).*



# Analysis (Requirements): Actors, Goals, System Boundaries



# NextGen POS: Process Sale Use Case Description

Use case UC1: Process Sale

Primary Actor: Cashier

Stakeholders and Interests:

-Cashier: Wants accurate and fast entry, no payment errors, …

-Salesperson: Wants sales commissions updated.

…

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions):

-Sale is saved. Tax correctly calculated.

…

Main success scenario (or basic flow): [see next slide]

Extensions (or alternative flows): [see next slide]

Special requirements: Touch screen UI, …

Technology and Data Variations List:

-Identifier entered by bar code scanner,…

Open issues: What are the tax law variations? …

Main success scenario (or basic flow):

The Customer arrives at a POS checkout with items to purchase.

The cashier records the identifier for each item. If there is more than one of the same item, the Cashier can enter the quantity as well.

The system determines the item price and adds the item information to the running sales transaction. The description and the price of the current

item are presented.

On completion of item entry, the Cashier indicates to the POS system that item entry is complete.

The System calculates and presents the sale total.

The Cashier tells the customer the total.

The Customer gives a cash payment ("cash tendered") possibly greater

than the sale total.

Extensions (or alternative flows):

If invalid identifier entered. Indicate error.

If customer didn't have enough cash, cancel sales transaction.

# Next Gen POS Use Case Diagram

UC1: ***Process Sale***

…

Main Success Scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.

…

7. Customer pays and System handles payment

**Extensions:**

7b. Paying by credit: Include *Handle Credit Payment.*

7c. Paying by check: Include *Handle Check Payment*

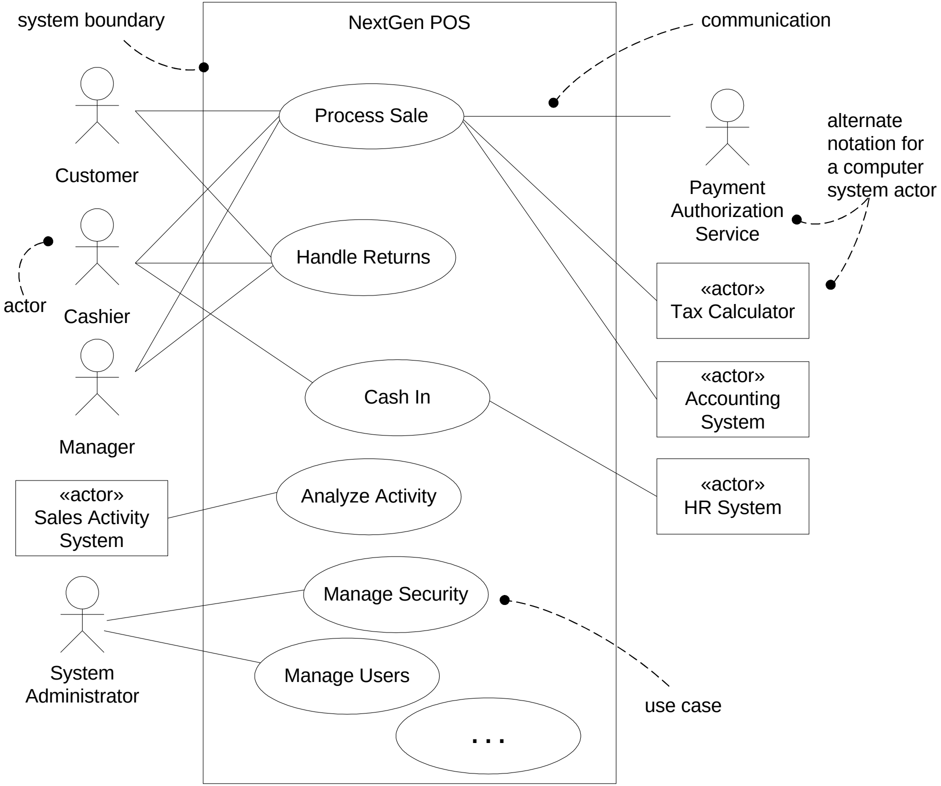
UC7: Process Rental

…

**Extensions:**

6b. Paying by credit: *Handle Credit Payment.*

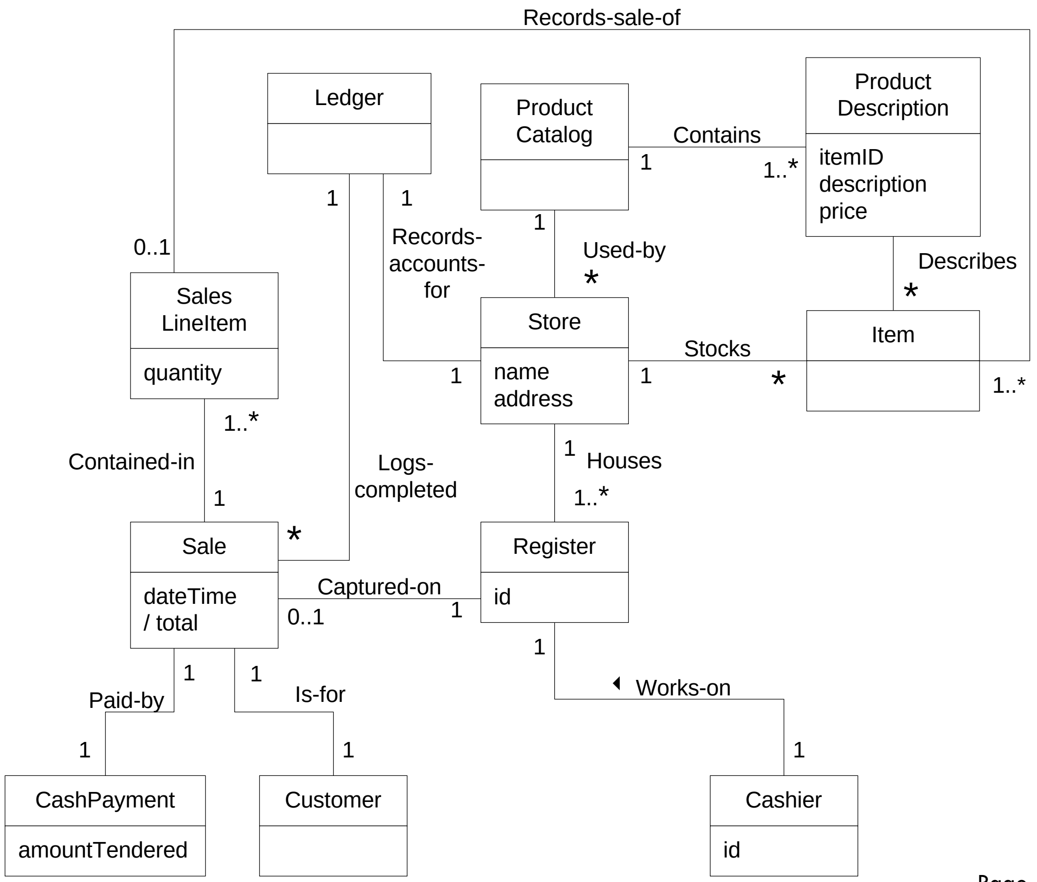
…



# NextGen POS Analysis: Domain Model

A conceptual perspective model
Partial domain model drawn with UML class diagram

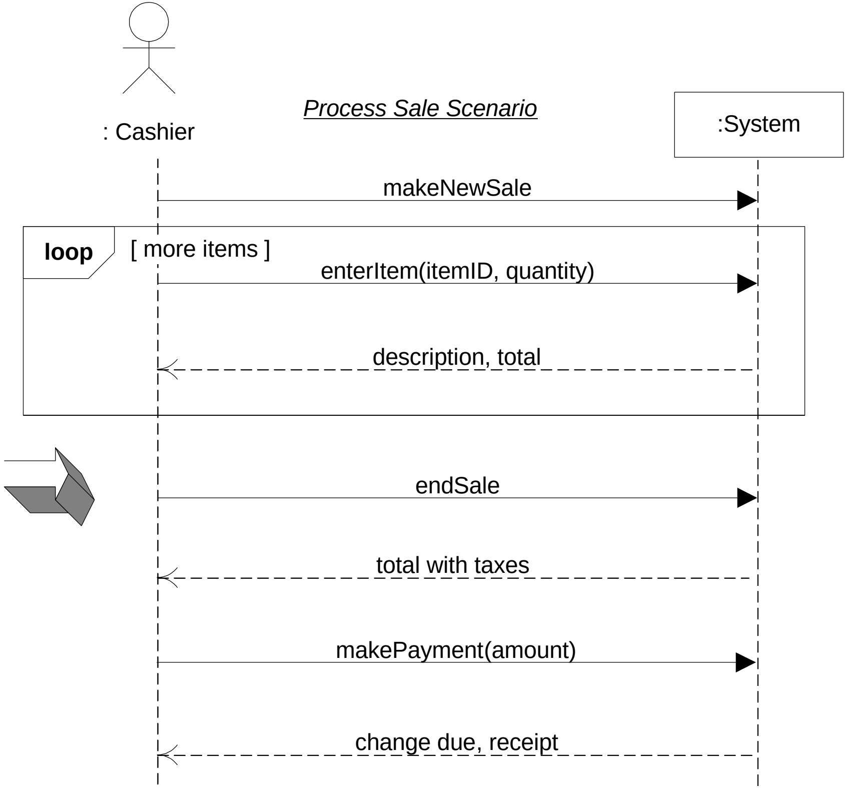
It shows conceptual classes with key associations



# NextGen POS Analysis: System Sequence Diagram (SSD)

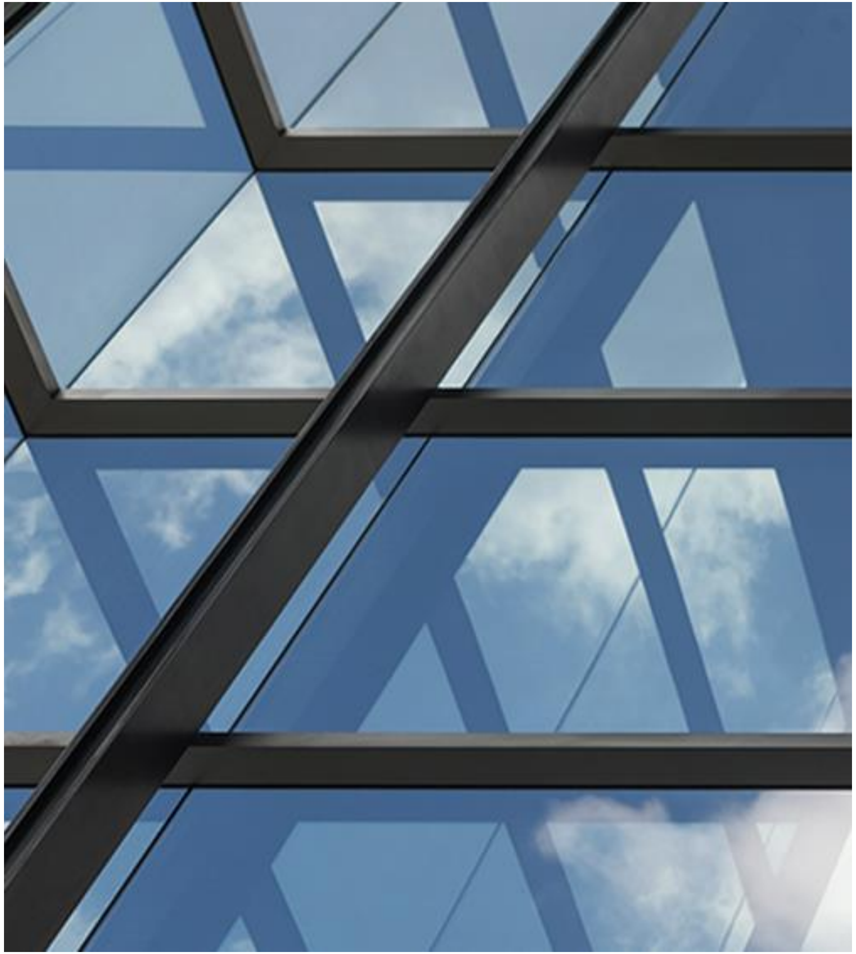
Simple cash-only *Process Sale* scenario:

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. System records sale line item and presents item description, price, 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 and System handles payment.
...



# NextGen POS Case Study: Design

OO Design with UML



THE UNIVERSITY OF SYDNEY

# Next Gen POS: From Analysis to Design

**Requirements Analysis (OOA)**

Business modelling – domain models

Use case diagrams

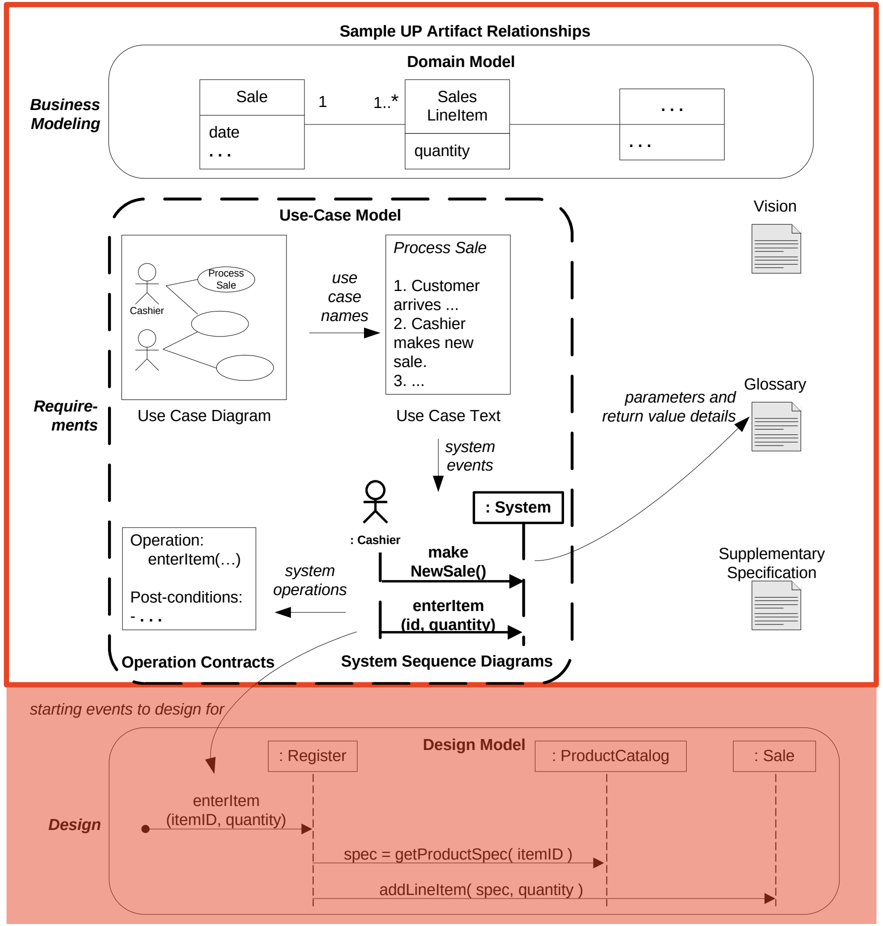
Use case description

System Sequence Diagrams

**Design (OOD)**

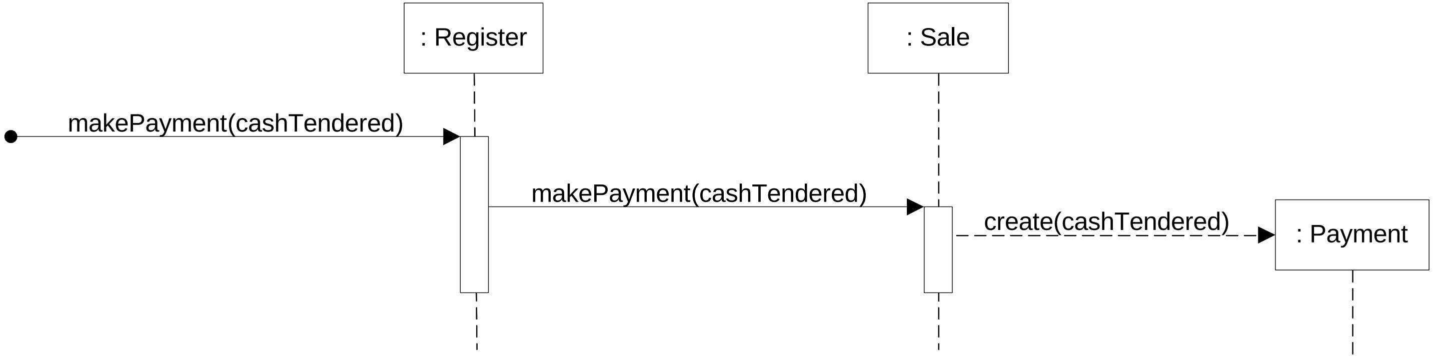
Sequence diagrams

Class diagrams

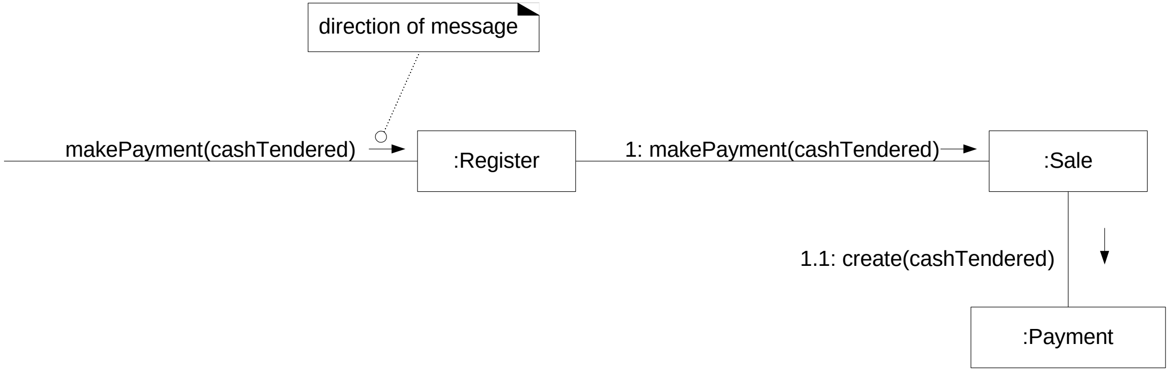


# NextGen POS: Interaction Diagrams

## Sequence Diagram

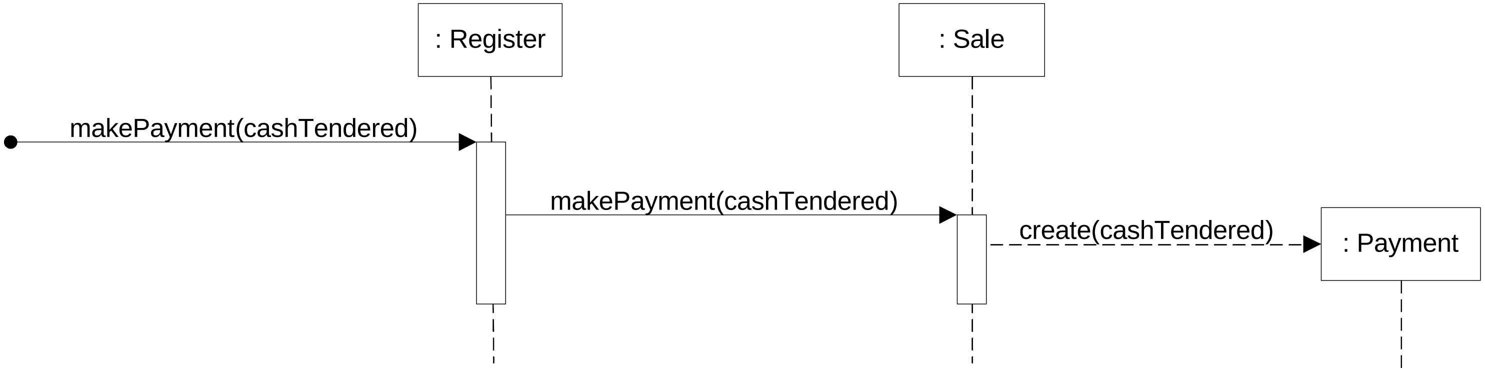


## Communication Diagram



direction of message
makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
:Sale
1.1: create(cashTendered)
:Payment

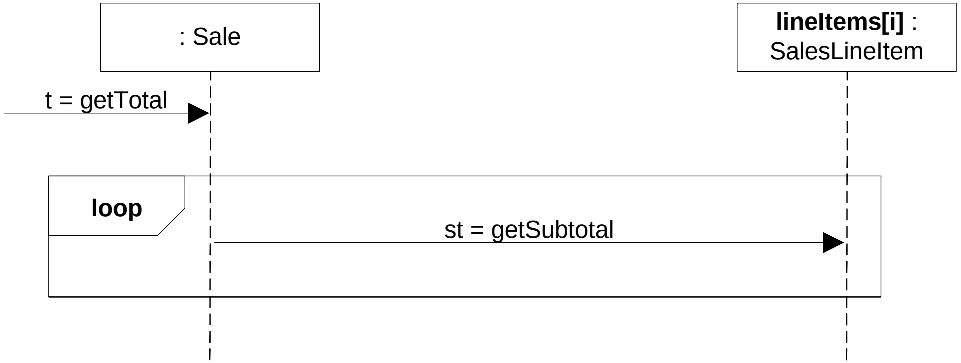
# NextGen POS: Sequence Diagrams

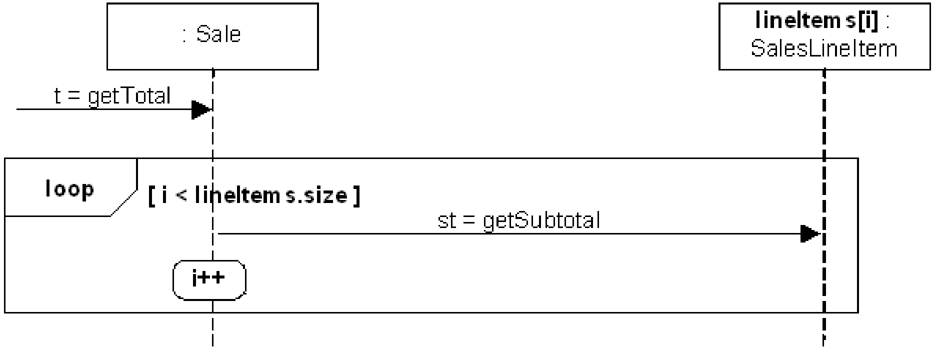


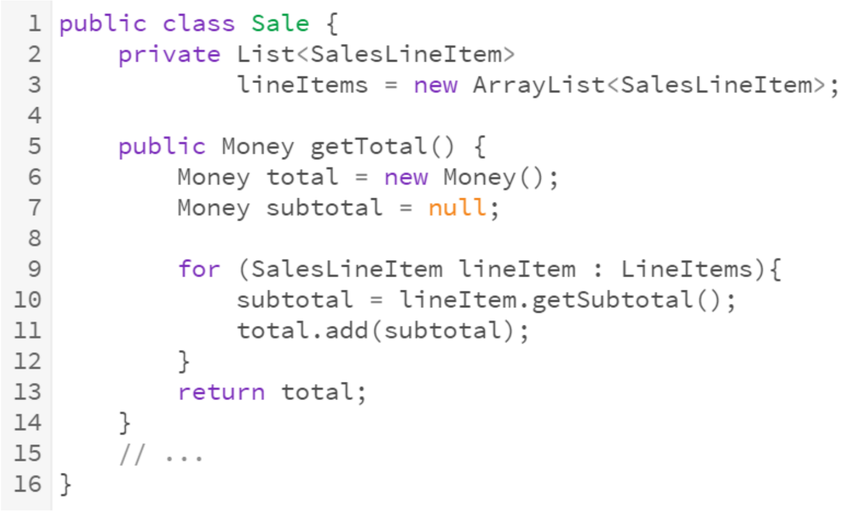
1. The message *makePayment* is sent to an instance of a *Register*. The sender is not identified
2. The *Register* instance sends the *makePayment* message to a *Sale* instance.
3. The *Sale* instance creates an instance of a *Payment.*

*How the skeleton of the Sale class should look like?*

# NextGen POS: Sequence Diagram (Iteration)

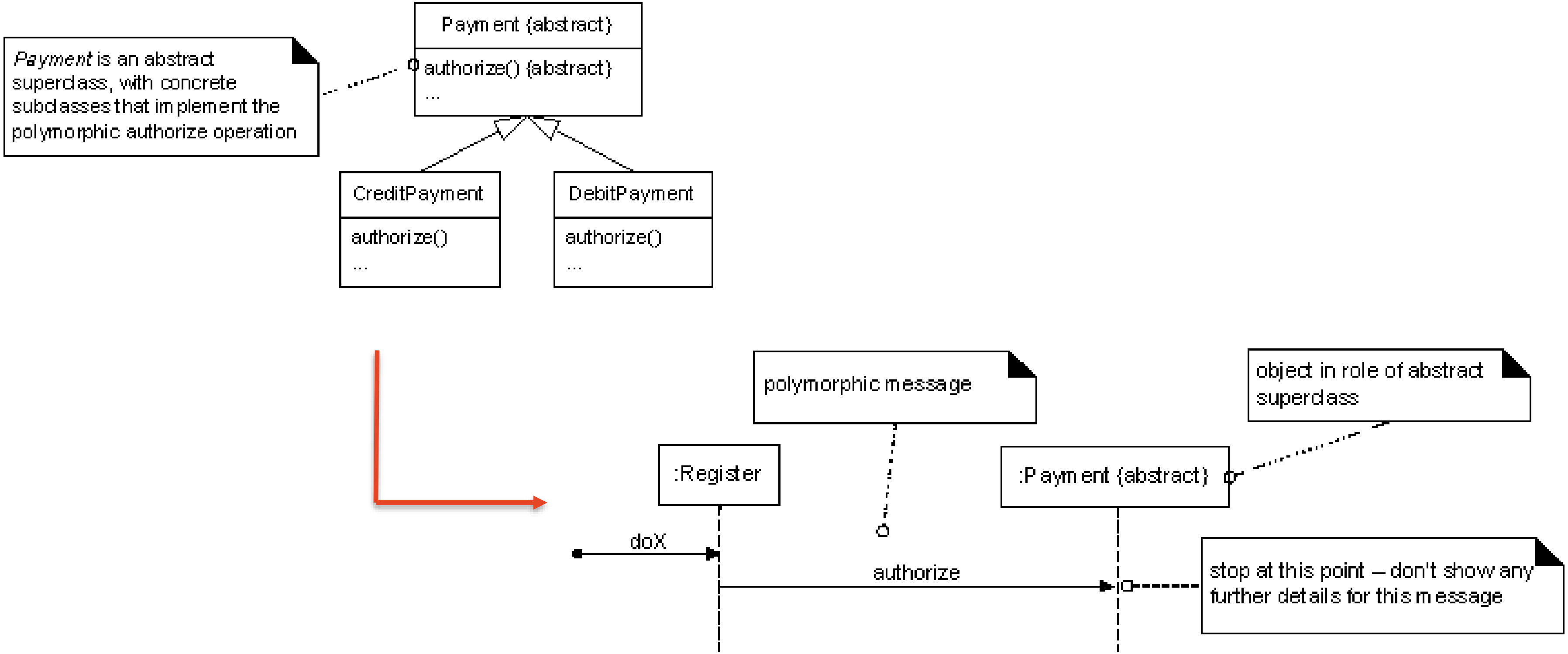




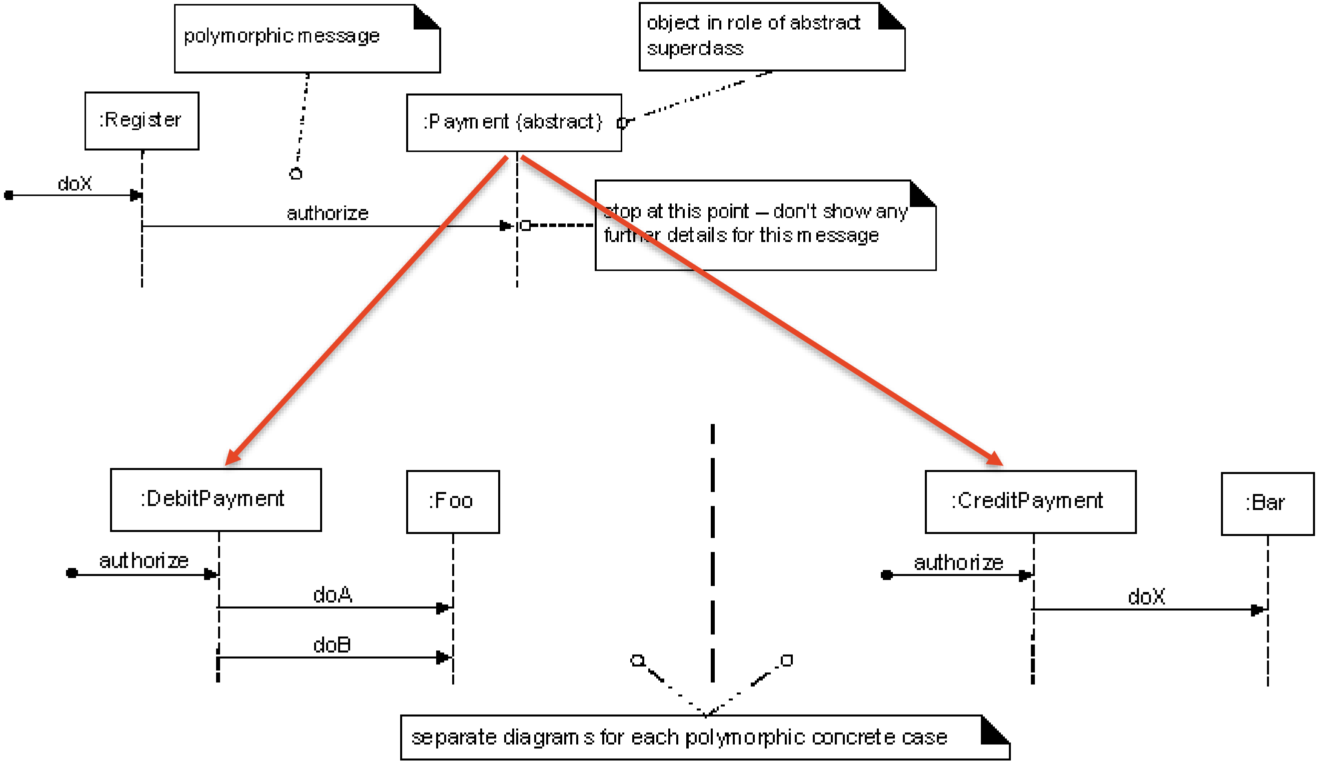


```
public class Sale {
    private List<SalesLineItem>
            lineItems = new ArrayList<SalesLineItem>;

    public Money getTotal() {
        Money total = new Money();
        Money subtotal = null;

        for (SalesLineItem lineItem : LineItems){
            subtotal = lineItem.getSubtotal();
            total.add(subtotal);
        }
        return total;
    }
    // ...
}
```

# NextGen POS: Polymorphic Messages


*Payment* is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation
Payment {abstract}
authorize() {abstract}
...
CreditPayment
authorize()
...
DebitPayment
authorize()
...
polymorphic message
object in role of abstract superclass
:Register
:Payment {abstract}
doX
authorize
stop at this point – don't show any further details for this message

# NextGen POS: Polymorphic Messages (Cont.)



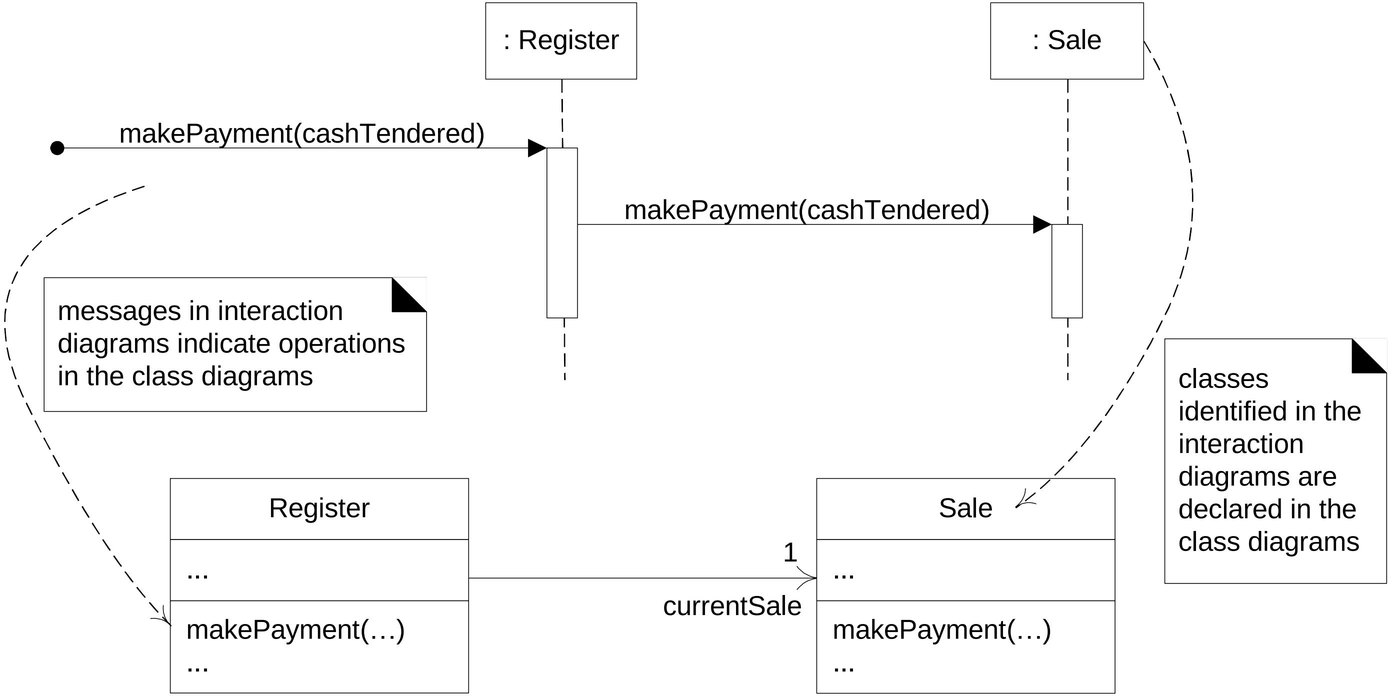
polymorphic message

object in role of abstract superclass

stop at this point – don't show any further details for this message

separate diagrams for each polymorphic concrete case

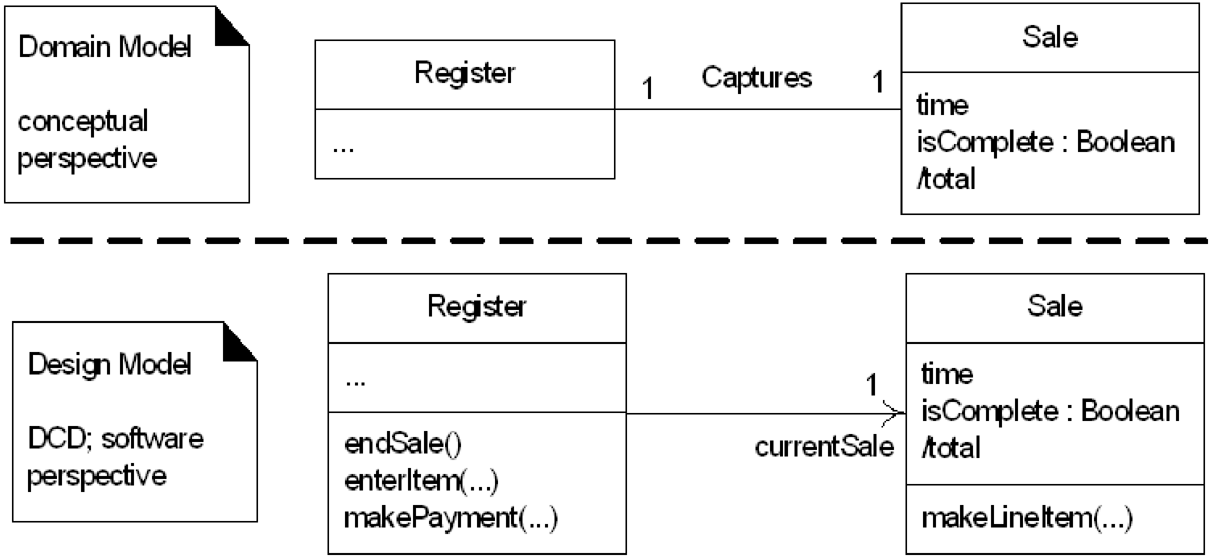
# NextGen POS: Interaction and Class Diagrams



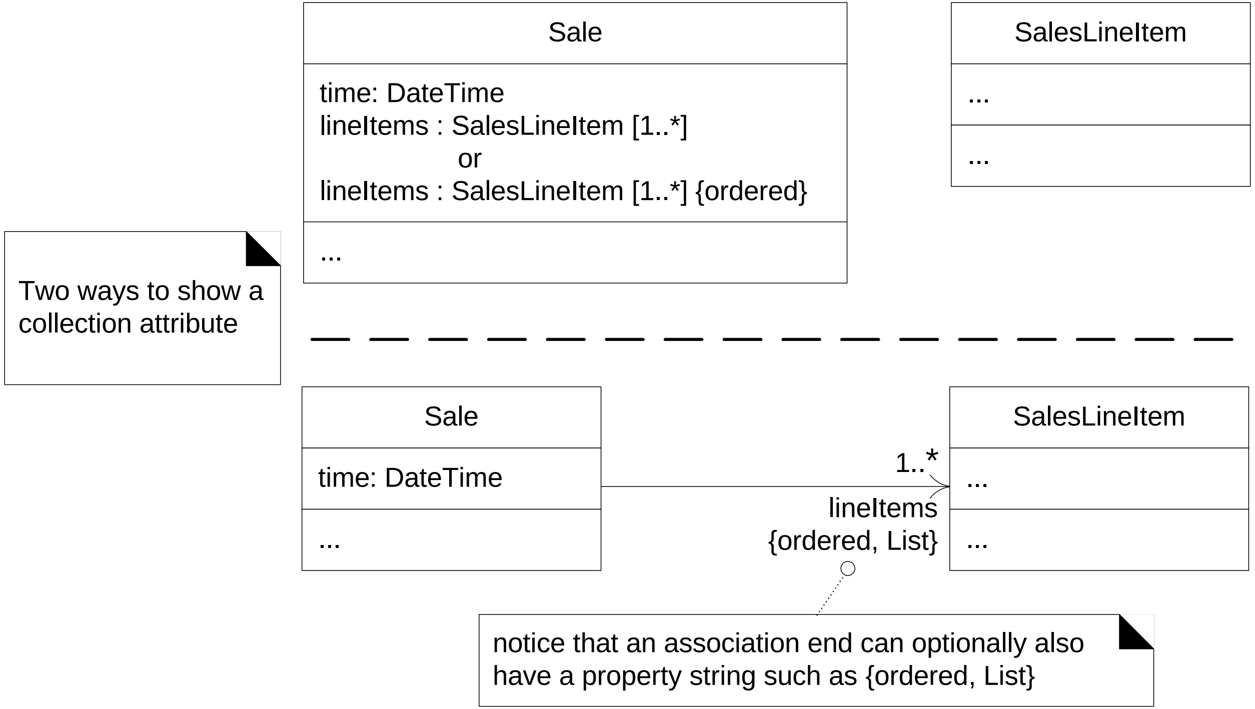
messages in interaction diagrams indicate operations in the class diagrams

classes identified in the interaction diagrams are declared in the class diagrams

# NextGen POS: Design Class Diagram



# NextGen POS: Collection of Attributes



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}

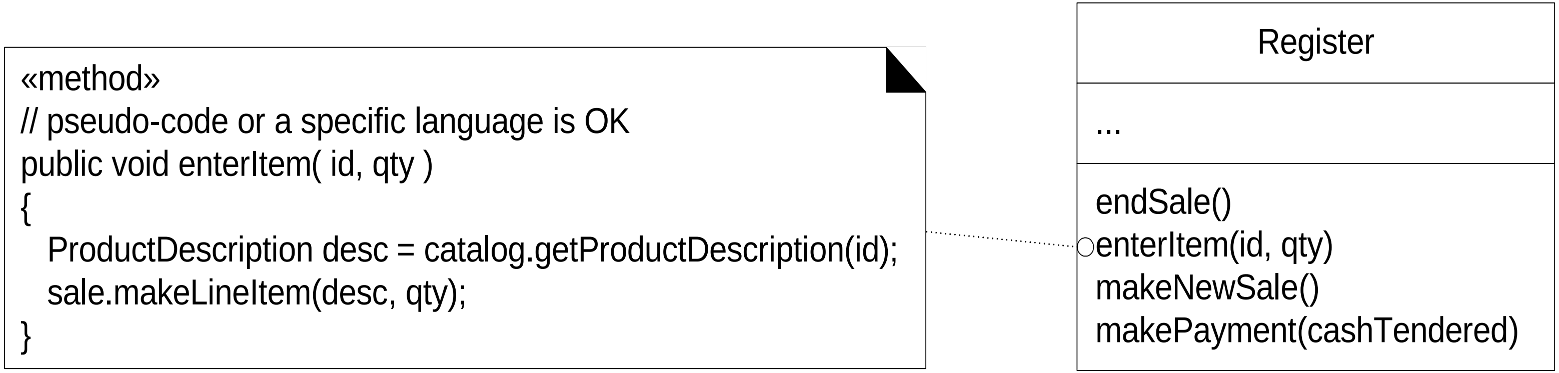
SalesLineItem

...

...

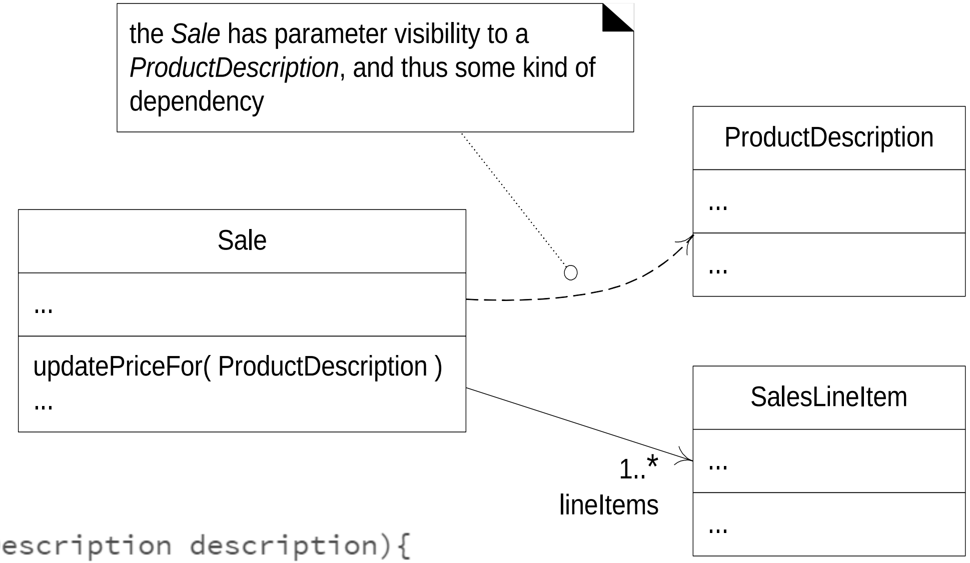
notice that an association end can optionally also have a property string such as {ordered, List}

# NextGen POS: Methods

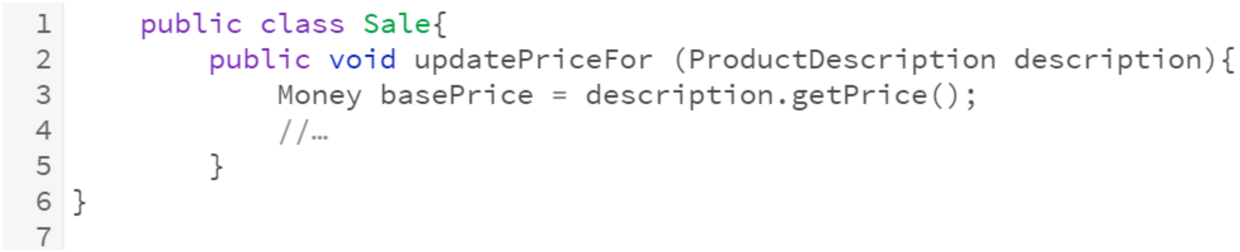


```
«method»
// pseudo-code or a specific language is OK
public void enterItem( id, qty )
{
   ProductDescription desc = catalog.getProductDescription(id);
   sale.makeLineItem(desc, qty);
}
```

# NextGen POS: Dependency



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

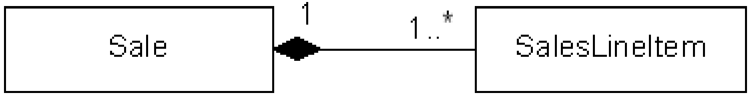


```
    public class Sale{
        public void updatePriceFor (ProductDescription description){
            Money basePrice = description.getPrice();
            //…
        }
}
```

# Next Gen POS: Composite Aggregation

*SalesLineItem* instance can only be part of one composite (*Sale*) at a time.

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



Composition (or Composite Aggregation is a strong kind of whole-part aggregation.
Use composition over aggregation as the latter was deemed by UML creators as “*Placebo*”

# References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.