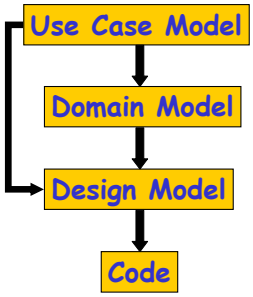
Domain Modeling

Relationships Among Models



Domain Modeling

- Idea: identify the important concepts in the problem domain
  - These concepts later will serve as basis for the design and the implementation
- Domain modeling (domain analysis)
  - We will consider object-oriented domain modeling in the context of the Unified Process

Objectives

- Learn to represent domain models
  - UML class diagram
- Learn how to identify conceptual classes and their attributes
- Learn about associations between classes
- Learn about generalization
- Experience with small domain models
  - in class and in assignments

The Domain Model

- Representation of real-world conceptual classes in the problem domain
  - With class attributes
- Representation of relationships between conceptual classes
  - Associations between classes
  - Generalization relationships
- Represented by a UML class diagram
  - But it could also be described in text

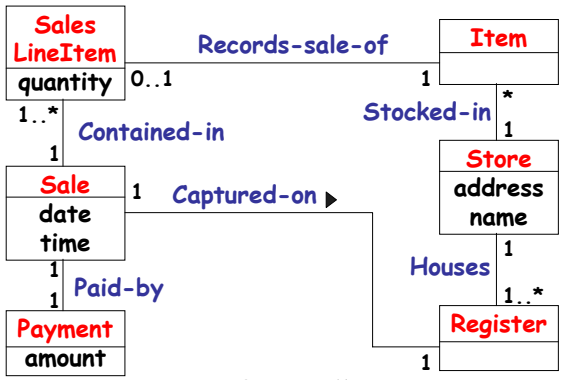
Models of Domain Concepts

Store
address
name

```
class Store {
  String address;
  String name;
  …
}
```

Of course, it is not always this simple …

A Class Diagram



UML Diagrams

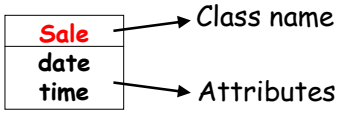
- UML is just notation
- Different diagrams mean different things in different contexts
- Conceptual perspective: description of the problem domain
- Specification perspective: description of software abstractions or components
  - e.g., no commitment to a particular language
- Implementation perspective

UML Class Diagrams

- Domain analysis: conceptual perspective
  - The elements are conceptual classes
- For design: specification or implementation perspective
  - The elements are design classes (software components with defined interfaces) or implementation classes (e.g. Java classes)
- The same kinds of diagrams are used for several purposes

Conceptual Classes

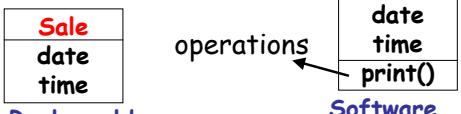
- Abstractions of concepts from the problem domain
  - Concepts such as Sale, Register, Item, ..
- UML representation



  - only attributes relevant to the problem

Focus on the Problem

- Do not represent software artifacts
  - Eventually will be the "inspiration" for software classes



Real-world concept in the problem domain
- no operations

Software class
- not part of the domain model
- created during design

Building the Domain Model

- Over several iterations during elaboration
- Driven by the use cases
  - In each iteration, the use case model is enriched, and the domain model is extended accordingly

Common Categories of Classes

Category	Examples
Physical objects	Register, Airplane
Places	Store, Airport
Transactions	Sale, Payment, Reservation
Roles of people	Cashier, Manager
Scheduled Events	Meeting, Flight
Records	Receipt, Ledger
Specifications and descriptions	FlightDescription, ProductSpecification
Catalogs of descriptions	ProductCatalog

Identifying Conceptual Classes

- Consider common categories
  - The list on the previous slide
- Identify nouns and noun phrases from the fully dressed use case
- Use analysis patterns: existing partial domain models created by experts
  - "recipes" for well-known problems and domains (e.g. accounting, stock market, …)

Example: Simplified "Process Sale"

Simplified scenario in **Process Sale**. No credit cards, no taxes, no external accounting system, no external inventory system, …

1. Customer arrives with goods
2. Cashier starts a new sale

Possible conceptual classes: **Customer**, **Cashier**, **Item** (-> goods), **Sale**

Example (cont)

3. Cashier enters item ID
4. System records sale line item and presents item description, price, and running total
5. At the end, Cashier tells Customer the total and asks for payment

Possible conceptual classes: **SalesLineItem**, **ProductSpecification** (description + price + item ID), **Payment**

- item ID, description, price, total: probably too simple to be separate classes

Example (cont)

6. Cashier enters amount tendered (cash)
7. System presents change due, and releases cash drawer
8. Cashier deposits cash and returns change
9. System presents receipt

Possible conceptual classes: **Register** (implied by cash drawer), **Receipt**

- amount, change: probably too simple

Example (cont)

- Want a completely integrated system
  - **Store**: has the items and the registers
  - **ProductCatalog**: stores the product specifications for all items
  - **Manager**: for example, starts all the registers in the morning
    - Need this for the initial implementation: to be able to start up the system
- There is no "correct solution"
  - Somewhat arbitrary collection of concepts

Possible Initial Model

- Just the conceptual classes
  - Attributes and associations later
- For this particular simplified scenario
  - Will evolve as more scenarios are explored

Register	Store	Item	Sale
Customer	Cashier	Manager	Payment
Product Specification	Product Catalog	Sales LineItem	

Decomposition of the Problem

- 70s and 80s: structured analysis and design
  - Focus: major functions
  - e.g. RecordRental, PayFines, etc.
- Since the 90s: object-oriented analysis and design
  - Focus: major concepts
  - e.g. Video, Customer, Cashier, etc.
- Similar to the difference between procedural and object-oriented languages

A Common Mistake

- Often things that are presented as attributes should be presented as conceptual classes
- Rule of thumb: if you cannot think of X as a number or text in the real world, X should probably be a conceptual class

Sale
store

OR ..?

Sale	Store
	address

A Common Mistake

- Another example

Flight
destination

OR ..?

Flight	Airport
	name

- If in doubt, make it a conceptual class
  - Attributes should be fairly rare in a domain model

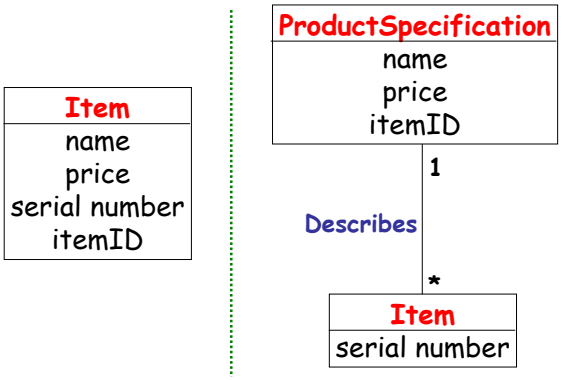
Specification Classes

- Example: class Item represents a physical item in a store
  - Each item has a unique serial number
  - All items of the same kind (e.g. JVC XV-S400 DVD player) have the same item ID and price
- We could represent ID and price as attributes of Item
  - But: suppose that we sell all items of a particular kind; we won't know the price
  - Also: unnecessary duplication of data

Specification Conceptual Classes

- In this case we need a separate conceptual class that is a description (a specification) of items
  - e.g. class **ProductSpecification**
- An instance of this class represents a description of information about items
  - Even if we sell all JVC XV-S400 DVD players, we still have information about their price/item ID

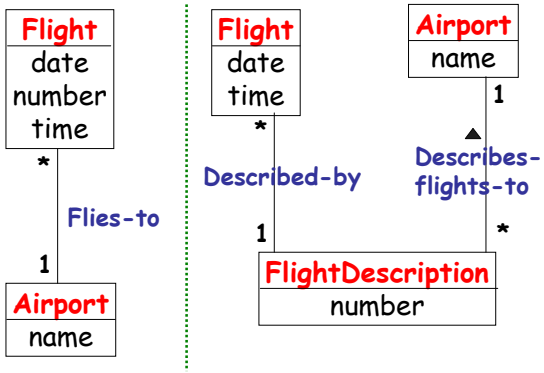
The Two Alternatives



When Do We Need This?

- Need description of an item or a service
  - Independent of the current existence of any instances of those items or services
- When specification classes would reduce redundant or duplicate information
  - e.g. many instances of the class have the same values for some attributes
- If the description alone can be in interesting relationships
  - e.g. all JVC XV-S400 DVD players are on sale until Dec 26th

Another Example



Associations in the Domain Model

- Relationship between instances of conceptual classes
  - "connectedness" between instances
  - e.g. an order is related to the customer that placed that order
- Think of it as a mathematical relation
  - Typically a binary relation: $R \subseteq S1 \times S2$
  - S1 = set of instances of the first class
  - S2 = set of instances of the second class

Associations in the Domain Model

- The relation changes with time
  - For any pair $(o1,o2) \in S1 \times S2$: at some moments of time the link exists, at other moments it doesn't
- An association typically represents a relatively permanent relationship
  - Often for the duration of the entire lifetime of the instance(s)
  - e.g. a sale is permanently associated with the register that captures it

UML Notation

- Named to enhance understanding of the relationship
- Multiplicity: what number of instances can be associated?
- Direction arrow: just helps the reader
  - No meaning for the model; often omitted

Multiplicity

- One instance of Store can be associated with zero or more Item instances

- Multiplicity at a particular moment
  - A man may be married to many women during his lifetime, but at any particular moment he is married to zero or one (hopefully)
  - Think of R ⊆ S1 × S2 at a particular moment

Representing Multiplicity

- Range: x..y
- Common notation for ranges
  - x..x -> x
  - x..infinity -> x..*
  - 0..infinity -> *
- Combination of ranges
  - x..y, z..w
  - e.g. "2,4" -> number of doors in a car
- Most common multiplicities: *, 1..*, 0..1, 1

Multiplicity Depends on the Viewpoint

- E.g. an item may be sold or discarded
  - If the requirements do not require tracking of such "strange" items, we can reflect this in the domain model
- Multiplicities may encode relevant domain constraints
  - But: it is not always clear

Typical Associations

- A is a physical/logical part of B
  - Wing-Airplane, SalesLineItem-Sale, FlightLeg-FlightRoute, Finger-Hand
- A is physically/logically contained in B
  - Item-Shelf, Passenger-Airplane, Flight-FlightSchedule
- A is recorded/reported/captured in B
  - Sale-Register, Reservation-FlightManifest
- A is a description of B
  - ProductSpecification-Item

Typical Associations

- A is a member of B
  - Cashier-Store, Pilot-Airline
- A uses or manages B
  - Cashier-Register, Pilot-Airplane
- A is related to a transaction B
  - Customer-Payment, Payment-Sale, Reservation-Cancellation
- A is owned by B
  - Airplane-Airline

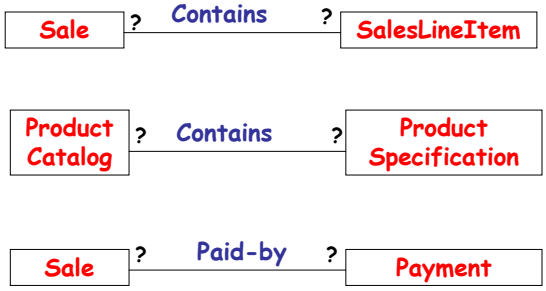
Finding Associations

- Consider the typical categories
- Focus on associations that are relevant with respect to the use cases
- SalesLineItem-Sale
  - A sale contains a set of line items
    - Permanent "whole-part" relationship
  - Needed in the context of the Process Sale use case (for the total and for the receipt)

Examples

- ProductSpecification-ProductCatalog
  - "contained-in" relationship
  - Given an item id, the system needs to look up the item description in the catalog
- Payment-Sale
  - Two related transactions: the payment is with respect to a particular sale
  - The payment info is needed to compute the change due

Examples



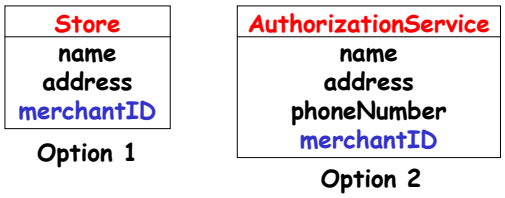
Somewhat More Complicated Example

- A store uses a set of external authorization services for payments

- Each service associates merchant ID with the store (different for each store)
  - The ID is provided by the store as part of the request for authorization
- A store has different merchant IDs for each service

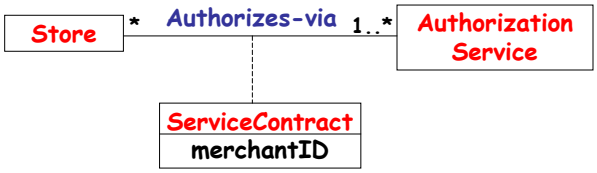
Stores and Services

- A software system at headquarters: many stores, many services
  - Where is the merchant ID located?



Association Class

- Attribute merchantID is conceptually related to the association, not to the individual classes
- Solution: association class
  - Represents attributes of the association



Association Classes

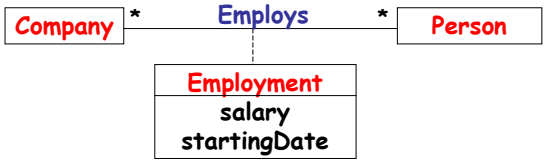
- An association class is a generalized form of an association
  - Association: set of pairs $(o1,o2) \in S1 \times S2$
  - Association class: set of pairs $(o1,o2) \in S1 \times S2$, where each pair has some attached info (attributes)
- The attributes of a pair may change with time (e.g., the merchant ID may change)
- Association classes may be associated with other classes (e.g. ternary relation)

When to Use Them?

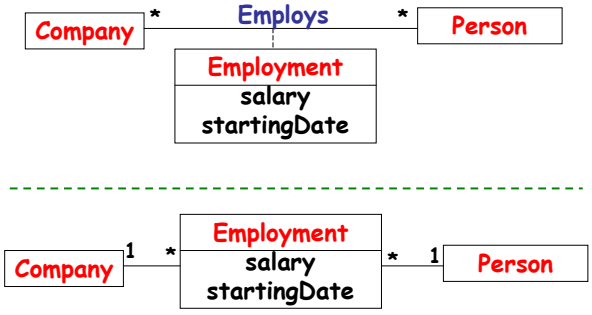
- When an attribute "doesn't fit" in the classes participating in an association
- When the lifetime of the attribute depends on the lifetime of the association
- Often used with many-to-many associations

Many-to-Many Association

- A company may employ several persons
- A person may be employed by several companies
  - Many people work two or even three jobs
- Attributes: salary, starting date, …



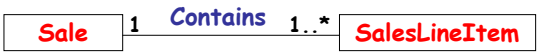
What is the Difference?



Associations and Their Implementation

- In the domain model: an association is conceptual and does not imply that a particular implementation will be used
  - Some domain-level associations may never be implemented
- In design and coding: there are standard mechanisms to implement the associations

Implementation Examples




```
class Sale {
   // set of references
   // to S.L.I. objects
   private Set items;
}
class SalesLineItem {
}
```

```
class Sale {
}
class SalesLineItem {
   private Sale encl_sale;
}
```

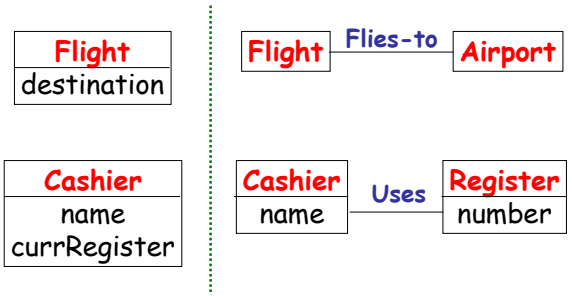
Could even be bi-directional: fields in both classes

Attributes in the Domain Model

- Attributes that are relevant for the scenarios under consideration
- Example: Process Sale use case
  - Need to know date/time of a sale to print a receipt and to log the sale
  - **Sale** needs attributes **date** and **time**

Attributes vs. Classes

- Attributes should not be complex domain concepts



Domain Model vs. Implementation

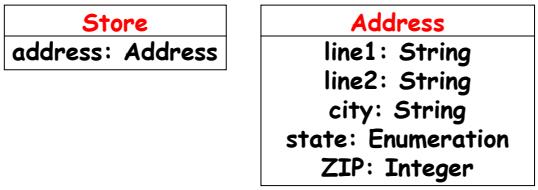
- **Key principle**: in the domain model, complex concepts should be related through associations, not through attributes
- In design/code, the **implementation** of the association may be through attributes of software classes
  - e.g. class Flight may have a field (attribute) that refers to an instance of Airport
  - But other implementations are also possible

Common Types of Attributes

- Primitive types: Number, String, Boolean
- Other simple types: Date, Time, Name, Address, Color, Phone Number, SSN, UPC (universal product code = barcode), ZIP, enumeration types, …
- Some simple attribute types (e.g. SSN) may need to be represented as separate conceptual classes

Simple Attribute Types as Classes

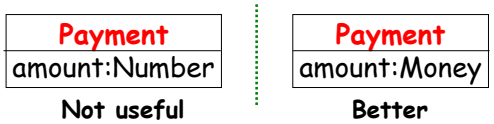
- The type has separate sections
  - e.g. address, phone number, name, item id



- The type has associated operations
  - e.g. parsing and validation for SSN

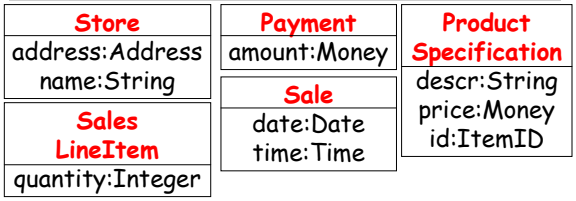
Simple Attribute Types as Classes

- Quantity with a unit
  - Most quantities have units: price, velocity, weight, etc.
    - Need to know the unit and to perform conversions
  - Represent different quantities as separate conceptual classes: Money, Weight, etc.



Not useful **Better**

"Process Sale" Use Case



- Store name/address: for receipt
- item id in Product Spec: for lookups
- Description/price in ProductSpec: for amount due and for display/receipt

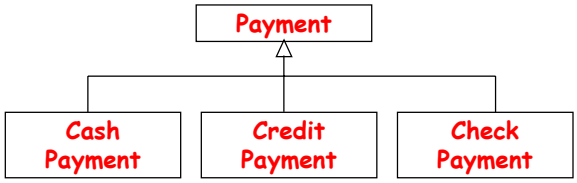
Entire domain model: Sec 12.9 of [Larman02]

Summary

- Conceptual classes
  - Special case: specification classes
- Attributes
  - Should be simple
- Associations: relationships that are relevant for the use cases
  - Multiplicity at a particular moment
  - Association classes

Generalization

- Superclass-subclass relationships
- Used in the **domain model** and in the **design model**

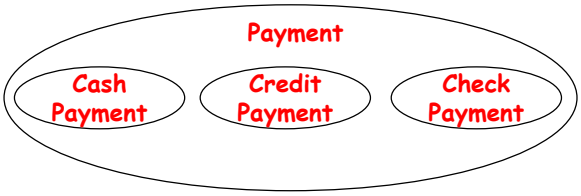


Basic Idea

- Domain model: a superclass represents a general concept, and a subclass represents some specialization
  - "CashPayment **is-a-kind-of** Payment"
- Design: the subclass interface conforms to the interface of the superclass
  - Software components with interfaces
  - The subclass can be used at any place where the superclass is allowed

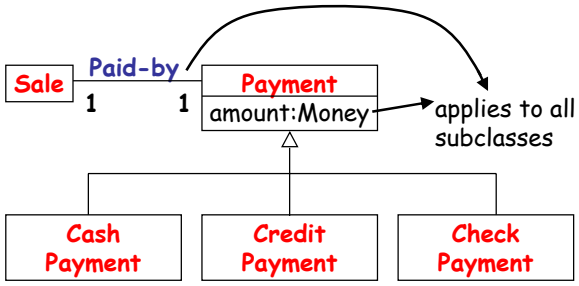
Meaning of Generalization

- All members of the subclass set are also members of the superclass set
  - "Every instance of CreditPayment is also an instance of Payment"
  - "is-a-kind-of"



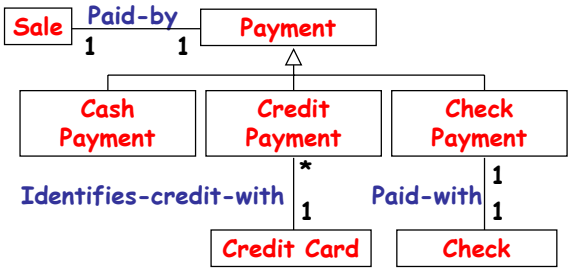
Meaning of Generalization

- All associations and attributes of the superclass apply to the subclass



Additions

- Subclasses could add associations and attributes



Motivation for Subclasses

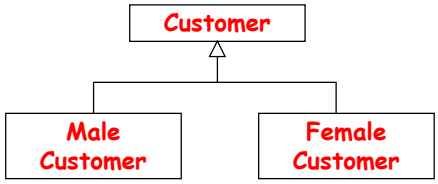
- Typical reasons for creating subclasses
- The subclass has additional attributes
  - e.g. in a library: superclass LoanableResource; subclass Book has attribute ISBN
- The subclass has additional associations
  - superclass LoanableResource; subclass Video is associated with Director

Motivation for Subclasses

- The subclass is handled/reacted to/ manipulated differently
  - superclass LoanableResource; subclass Software: requires a deposit
- If a "subconcept" (a subset of instances) of some existing class has some of these properties, the creation of a subclass should be considered
  - Unnecessary subclasses should be avoided

An Example

For the POS system, is this a good idea?



Motivation for Superclasses

- The model may contain a set of classes for which it makes sense to create a superclass
- When the classes represent variations of a similar concept
  - e.g. if we have CreditAuthorizationService and CheckAuthorizationService, it may be a good idea to create superclass AuthorizationService
  - Duplicate attributes/associations often indicate that a superclass is needed

Creating Superclasses

- When creating a new superclass, always need to make sure that the relationship is "is-a-kind-of"
  - and all superclass attributes/associations apply to all subclasses
- If all subclasses have the same attribute, it should be moved to the superclass
- If all subclasses have the same association, it should also be moved

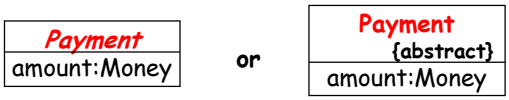
Example

- POS system uses external authorization services for credit payments
- Three different kinds of payment transactions: requests, approvals, denials
- Each transaction has date and time associated with it
- Approvals and denials have processing time associated with them
  - e.g. for performance measurements

Abstract Conceptual Classes

- Any instance of the class must be an instance of some known subclass

- UML notation

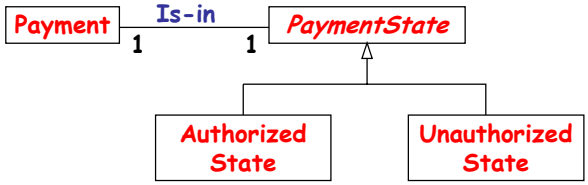


Modeling of State

- Do not model the states of concept X as subclasses of X
  - e.g. state of payment = { unauthorized, authorized }
  - Do not create subclasses AuthorizedPayment and UnauthorizedPayment
- Problem: an instance of the concept may change states
  - a payment is initially in unauthorized state, and then moves to authorized state

Modeling of State

- One possible solution: create a state hierarchy, and associate it with X



Example of the *State* design pattern

Inheritance

- Typically generalization is implemented through inheritance

```
class Payment { … }
class CreditPayment extends Payment { … }
```

- The subclass inherits all methods and fields in the superclass
- The subclass may add new fields and methods, and may redefine methods inherited from the superclass

Summary of Domain Modeling

- Central focus: conceptual classes
  - Plus their associations, attributes, and generalization relationships
  - Represented by an UML class diagram
- No single correct model
  - All models are approximations of the domain
  - Capture essential domain aspects

UP Artifacts

Artifact	Incep	Elab	Const	Trans
Use-Case Model	X	X		
Supplem. Spec	X	X		
Domain Model		X		
Design Model		X	X	
Implem. Model		X	X	X

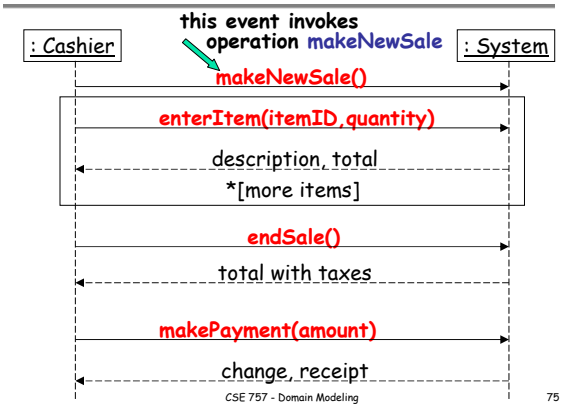
Requirements analysis: **Use-Case Model + Supplementary Specification**
Domain analysis: **Domain Model**
Design: **Design Model**
Coding: **Implementation Model**

Operation Contracts

More on the Use Case Model

- **Use cases:** primary way to describe requirements
  - For certain scenarios: **systems sequence diagrams**
- Use Case Model and Domain Model: developed in parallel
- **Operation contracts**: a way to describe in more detail the use cases
  - Part of the Use Case Model

System Operations



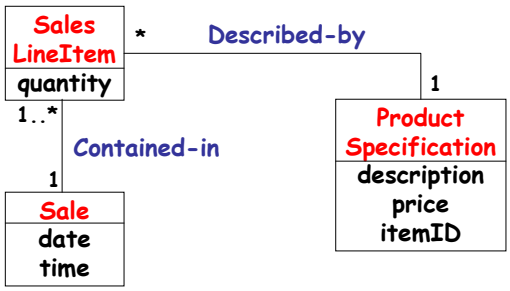
Operation Contracts

- Semantics of system operations in terms of state changes to objects in the Domain Model
  - Only for more complex and subtle system operations
  - Level of detail that clarifies what complex operations should do

Example of a Contract

- enterItem(*itemID*:ItemID,*quantity*:int)
- Cross References: Process Sale use case
- Preconditions: there is a sale underway
- Postconditions (state changes)
  - A **SalesLineItem** instance sli is created
  - sli is associated with the current Sale
  - sli.quantity becomes *quantity*
  - sli is associated with a ProductSpecification, based on *itemID* match

Relevant Part of the Domain Model



Other Examples

- Stand-alone system
  - No external inventory system
- Adding items to the inventory
  - An instance of Item is created
  - The new instance is associated with the Store
- Removing items from the inventory
  - The association is destroyed and the Item instance is also destroyed