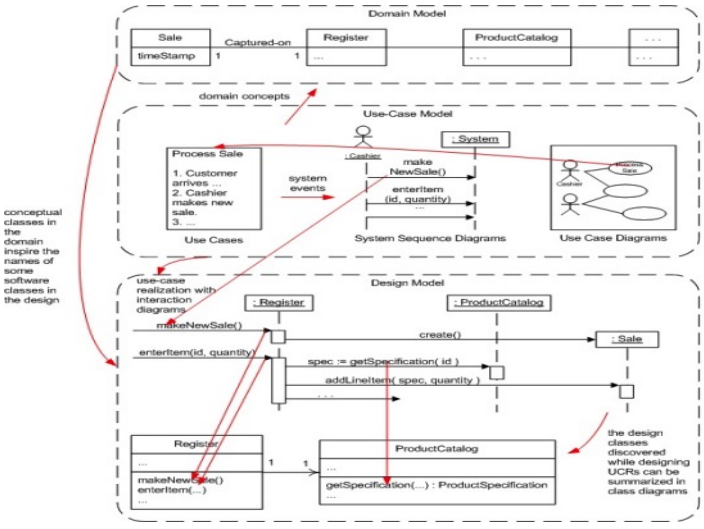
CASE STUDY: THE NEXTGEN POS SYSTEM

- The case study is the NextGen point-of-sale (POS) system.
- This is straightforward problem domain. Analysis the requirement and design problems to solve. In addition, POS systems using object technologies.
- A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store.
- It includes hardware components such as a computer and bar code scanner, and software to run the system.
- It interfaces to various service applications, such as a third-party tax calculator and inventory control.
- These systems relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).
- A POS system increasingly must support multiple and varied client-side terminals and interfaces.
- These include a thin-client Web browser terminal, a regular personal computer with Java Swing graphical user interface, touch screen input, a commercial POS system .
- Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added.
- Need a mechanism to provide this flexibility and customization.
- Using an iterative development strategy, we are gain wireless PDAs, and so forth.
- Creating g to proceed through requirements, object-oriented analysis, design, and implementation.
- **Inception**
    - Inception is the initial step to formulate the product vision - the core requirements - in terms of functionality, scope, performance, capacity, technology base.
    - Success criteria
    - An initial risk assessment.
    - An estimate of the resources required to complete the elaboration phase.

      Purpose of inception
        - Collect enough information to establish a common vision
        - Deciding if moving forward is feasible.

      Use cases
- The use cases are text stories used to discover and record requirements.
- use cases influence many aspects of a project.
- They are input to artifacts.
- It also influences
    - Analysis
    - Design
    - Implementation
    - Project management
    - Test artifacts.



Definitions

- Actor
  Something with behavior
  - A person
  - Computer system
  - Organization
- Scenario
  - A scenario is a specific sequence of actions and interactions between actors and system.
  - Ex scenario of successfully purchasing items with cash.
  - Scenario of failing to purchase items because of credit payment denial.
- Use case
  - Use case is a collection of related success and failure scenarios that describe an actor using the system to support a goal.

Ex main success scenario

A customer arrives at a shop to return item
the cashier uses POS system and record returned items

- Alternate scenario
  - If customer paid through credit card and if it return due to credit account is rejected, pay the amount with cash.
  - Thus the RUP defines the use case as
    - " A set of use case instances, where each instance is a sequence of actions to a system that performs an observable result of value to a particular actor."

Use cases and use case models

- Use cases are text documents and not diagrams.
- Use case modeling is primarily an act of writing text, not drawing diagrams.
- Use cases are functional requirements or behavior.
- Use cases emphasize the user goals and perspectives.
- Strength of use cases is the ability to scale both up and down in terms of
  - Sophistication and formality.

Three kinds of actors

- Actors are roles played not only by people and also by organization.

1. Primary actor-it has user goals fulfilled by using services of system under discussion.
    1. Cashier
2. Supporting actor – provide services to SUD
    1. Automated payment authorization.
3. Offstage actor – it has an interest in the behavior of use cases but not primary or supporting
    1. A government tax agency

Use case formats

1. Brief 2.Causal 3.Fully dressed

Brief –

- one paragraph summary of main success scenario
- It is written during early requirement analysis
- It is written to get a quick sense of subject and scope.
- It can be written in few minutes

  Ex Process sale
- **casual**—informal paragraph format. Multiple paragraphs that cover various scenarios.

  Example was casual.
- fully dressed—the most elaborate. All steps and variations are written **in** detail, and there are supporting sections, such as preconditions and success guarantees.

Example NextGen case study

Use cases Template

Use case section	**comment**
Name	Start with verb
Scope	The system under design
Level	User goal or sub function
Primary Actor	Calls on the system to deliver its service
Stakeholders and Interests	Who cares about this use case, and what do they want?
preconditions	What must be true on start and worth telling the reader?
Success Guarantee	What must be true on successful completion, and worth telling the reader?
Main Success Scenario	A typical, unconditional happy path scenario of success
Extensions	Alternate scenarios of success
Special requirement	Related non functional requirement

USE CASE : Process Sale

(FULLY DRESSED VERSION)

- **Primary Actor: Cashier**
- **Stakeholders and Interests:**

– Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages

are deducted from his/her salary.
– Salesperson: Wants sales commissions updated.
– Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
– Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
– Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
– Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

- **Preconditions: Cashier is identified and authenticated.**
- **Success Guarantee (post conditions): Sale is saved. Tax is correctly calculated.**
- Accounting and Inventory are updated. Commissions recorded. Receipt is generated.
- **Main Success Scenario (or Basic Flow):**
- 1. Customer arrives at 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.
- Price calculated from a set of price rules.
- 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.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

- **Extensions (or Alternative Flows):**

a. At any time, System fails:

- To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.
2. System reconstructs prior state.

2a. System detects anomalies preventing recovery:

1. System signals error to the Cashier, records the error, and enters a clean state.

2. Cashier starts a new sale.

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

3-6a: Customer asks Cashier to remove an item from the purchase:

1. Cashier enters item identifier for removal from sale.
2. System displays updated running total.

3-6b. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:

1. System records sale so that it is available for retrieval on any POS terminal.

4a. The system generated item price is not wanted (e.g., Customer complained about something and is offered a lower price):

1. Cashier enters override price.
2. System presents new price.

5a. System detects failure to communicate with external tax calculation system service:

1. System restarts the service on the POS node, and continues.

1a. System detects that the service does not restart.

1. System signals error.
2. Cashier may manually calculate and enter the tax, or cancel the sale.

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):

1. Cashier signals discount request.
2. Cashier enters Customer identification.
3. System presents discount total, based on discount rules.

5c. Customer says they have credit in their account, to apply to the sale:

1. Cashier signals credit request.
2. Cashier enters Customer identification.
3. Systems applies credit up to price=0, and reduces remaining credit.

6a. Customer says they intended to pay by cash but don't have enough cash:

1a. Customer uses an alternate payment method.

1b. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

7b. Paying by credit:

1. Customer enters their credit account information.
2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.

2a. System detects failure to collaborate with external system:

1. System signals error to Cashier.
2. Cashier asks Customer for alternate payment.
3. System receives payment approval and signals approval to Cashier.

3a. System receives payment denial:

1. System signals denial to Cashier.
2. Cashier asks Customer for alternate payment.

4. System records the credit payment, which includes the payment approval.
5. System presents credit payment signature input mechanism.
6. Cashier asks Customer for a credit payment signature. Customer enters signature.

7c. Paying by check...

7d. Paying by debit...:

7e. Customer presents coupons:

1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.

1a. Coupon entered is not for any purchased item:

1. System signals error to Cashier.

9a. There are product rebates:

1. System presents the rebate forms and rebate receipts for each item with a rebate.

9b. Customer requests gift receipt (no prices visible):

1. Cashier requests gift receipt and System presents it.

- **Special Requirements:**

– Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.

– Credit authorization response within 30 seconds 90% of the time.

– Somehow, we want robust recovery when access to remote services such the inventory system is failing.

– Language internationalization on the text displayed.

– Pluggable business rules to be insert able at steps 3 and 7.

Technology and Data Variations List:

3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.

3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

7a. Credit account information entered by card reader or keyboard.

7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

What are the tax law variations?

– Explore the remote service recovery issue.

– What customization is needed for different businesses?

– Must a cashier take their cash drawer when they log out?

– Can the customer directly use the card reader, or does the cashier have to do it?

Meaning of the sections

- Preface Elements

1.scope – it bounds the system under design

2. Since use cases describes how a system is used by its customer and partners.

The enterprise level description- business use case .

- Level
  - Use cases are classified as
    - User goal level
    - Sub function level
- **Primary Actor: The principal actor that calls upon system services to fulfill a goal.**
- **Stakeholders and Interests:**

– Cashier: Wants accurate, fast entry and no payment errors, as cash drawer shortages are deducted from his/her salary.

– Salesperson: Wants sales commissions updated.

Preconditions and Success Guarantees

- **Preconditions: Cashier is identified and authenticated.**
- **Success Guarantee (post conditions): Sale is saved. Tax is correctly calculated.**
- Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Main Success Scenario and Steps (or Basic Flow)

- This has also been called the “happy path” scenario, or the more prosaic “Basic Flow”.
- It describes the typical success path that satisfies the interests of the stakeholders.

It does *not include any conditions or branching*

- The scenario records the steps, of which there are three kinds:
- 1. An interaction between actors.3
- 2. A validation (usually by the system).
- 3. A state change by the system (for example, recording or modifying something).
- **Main Success Scenario:**
- 1. Customer arrives at a POS checkout with items to purchase.
- 2. Cashier starts a new sale.

- 3. Cashier enters item identifier.
- 4. ...
- Cashier repeats steps 3-4 until indicates done.
- 5. ...

Extensions (or Alternate Flows)

- Extensions are very important. They indicate all the other scenarios or branches, both success and failure.
- Observe in the fully dressed example that the Extensions section was considerably longer and more complex than the Main Success Scenario section.
- They are also known as "Alternative Flows."
- They are noted with respect to its steps 1..N
- **Extensions:**

3a. Invalid identifier:

- 1. System signals error and rejects entry.
- 3b. There are multiple of same item category and tracking unique item identity not
- important (e.g., 5 packages of veggie-burgers):
- 1. Cashier can enter item category identifier and the quantity.
- An extension has two parts: the condition and the handling.
- 3-6a: Customer asks Cashier to remove an item from the purchase:
- 1. Cashier enters the item identifier for removal from the sale.
- 2. System displays updated running total.
- 7b. Paying by credit:
- 1. Customer enters their credit account information.
- 2. System requests payment validation from external Payment Authorization Service
- System.
- 2a. System detects failure to collaborate with external system:
- 1. System signals error to Cashier.

2. Cashier asks Customer for alternate payment
3. If it is desirable to describe an extension condition as possible during any (or at least most) steps, the labels *a, *b, ..., can be used.
4. *a. At any time, System crashes:
5. In order to support recovery and correct accounting, ensure all transaction sensitive
6. state and events can be recovered at any step in the scenario.
7. 1. Cashier restarts the System, logs in, and requests recovery of prior state.
8. 2. System reconstructs prior state.

Special Requirements

- The non-functional requirement,
- Quality attribute,
- Constraint

relates specifically to a use case represent Special Requirements.

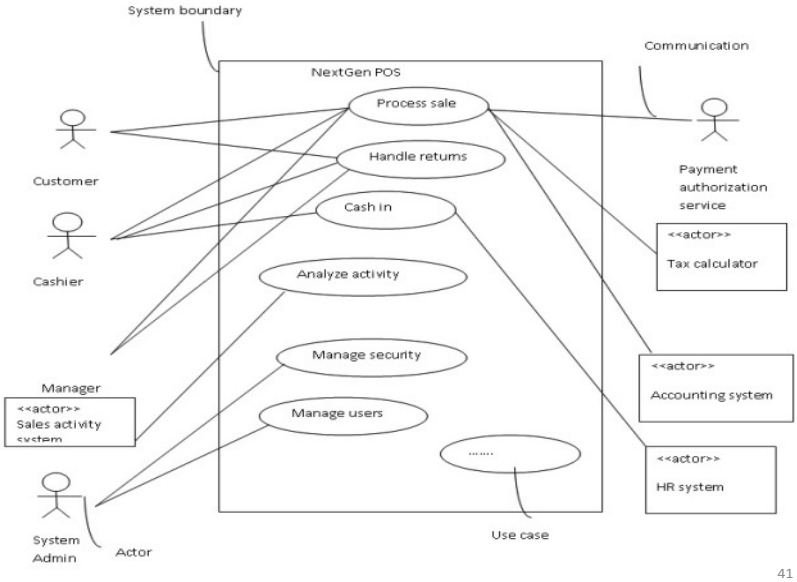
- These include qualities such as performance, reliability, and usability, and design constraints
- **Special Requirements:**
- – Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- – Credit authorization response within 30 seconds 90% of the time.
- – Language internationalization on the text displayed.
- – Pluggable business rules to be insertable at steps 2 and 6.

Technology and Data Variations List

- 3a. Item identifier entered by laser scanner or keyboard.

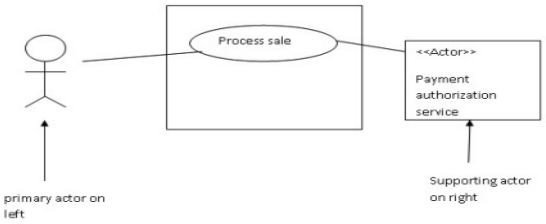
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Partial use case diagram – POS system



41

Some prefer to highlight external computer system actors with an alternate notation



Use Cases Within the UP

- Use cases are vital and central to the UP, which encourages **use-case driven development.**
- **This implies:** Requirements are primarily recorded in use cases (the Use-Case Model); other requirements techniques (such as functions lists) are secondary, if used at all.
- Use cases are an important part of iterative planning. The work of an iteration is—in part—defined by choosing some use case scenarios, or entire use cases. And use cases are a key input to estimation.

- **Use-case realizations drive the design. That is, the team designs collaborating** objects and subsystems in order to perform or realize the use cases. Use cases often influence the organization of user manuals.
- The UP distinguishes between system and business use cases. **System use cases are what have been examined in this chapter, such as *Process Sale. They*** are created in the Requirements workflow, and are part of the Use-Case Model.
- **Business use cases are less commonly written. If done, they are created in the**
- Business Modeling workflow as part of a large-scale business process engineering effort, or to help understand the context of a new system in the business.

Use Cases and Requirements Specification Across the Iterations

- Use Cases within Inception
  - Not all use cases are written in their fully dressed format during the inception phase.
  - Most of the interesting, complex, or risky use cases are written in brief format;

Use Cases within Elaboration

- This is a phase of multiple time boxed iterations (for example, four iterations) in which risky, high-value, or architecturally significant parts of the system are
- incrementally built . By the end of elaboration, "80-90%" of the use cases are written in detail.
- Use Cases within Construction
  - The construction step is composed of time boxed iterations (for example, 20 iterations of two weeks each) that focus on completing the system, once the risky and core unstable issues have settled down in elaboration.

Use Cases in the NextGen Inception Phase

Fully Dressed	Casual	Brief
Process Sale Handle Returns	Process Rental Analyze Sales Activity Manage Security	Cash In Cash Out Manage Users Start Up Shut Down Manage System Tables

Use case relationship

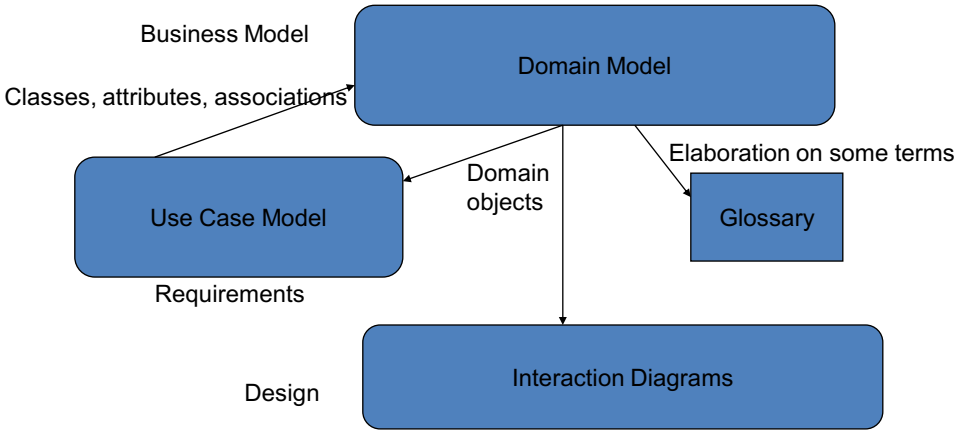
- Include
- Generalization
- Extend

Domain Model

Objectives

- Identify conceptual classes related to the current iteration.
- Create an initial domain model.
- Model appropriate attributes and associations.

Domain Model Relationships



Overview

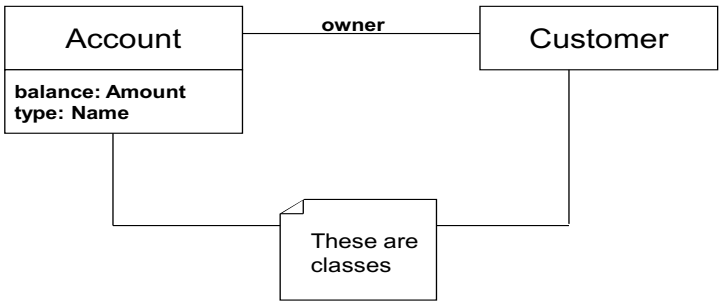
- **A domain model is the most important and classic model in OO analysis.**
- **A Domain Model visualizes noteworthy concepts or objects in the domain.**
- **You will be able to:**
  - **Read and write the UML class diagram notation for a Domain Model**
  - **Create a Domain Model**
  - **Apply guidelines**
  - **Relate it to other artifacts**
- DEFINITION & MOTIVATION: Domain Model
- **A *Domain Model* visualizes, using UML class diagram notation, noteworthy concepts or objects.**
  - **It is a kind of "visual dictionary."**
  - ***Not* a picture of software classes.**
- **It helps us identify, relate and visualize important information.**

Domain Model

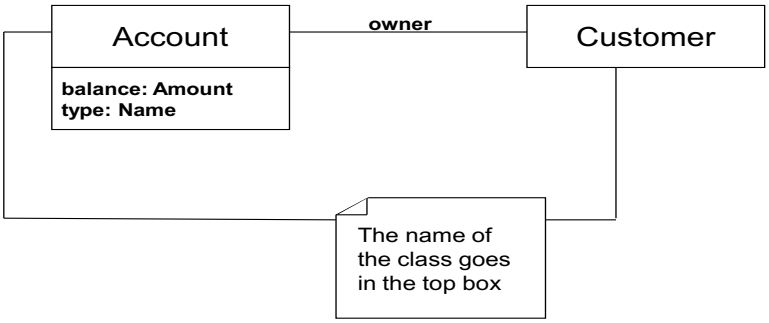
- Next, for each class, we determine the relevant information (attributes and relationships):
  - For Account:
    - Balance (Amount) (e.g. $1000.00)
    - Type (Name) (e.g. Credit)
    - Owner (Customer) (e.g. Barb Smith)
- Information that can be represented with atomic types are attributes
  - The first two examples above are attributes
- Information that is expressed in terms of other classes are relationships with those classes
  - We can call these relationships *associations* for now

A Partial Domain Class Diagram

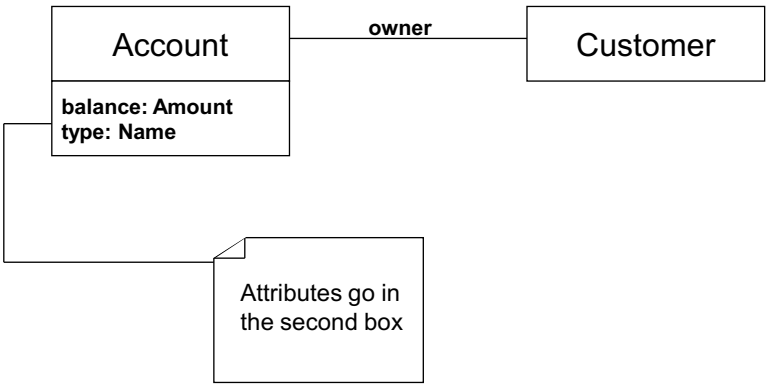
A Partial Domain Class Diagram



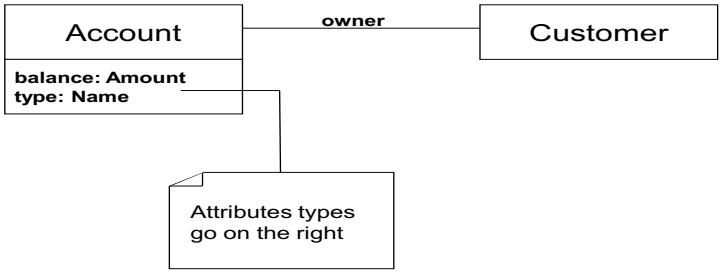
A Partial Domain Class Diagram



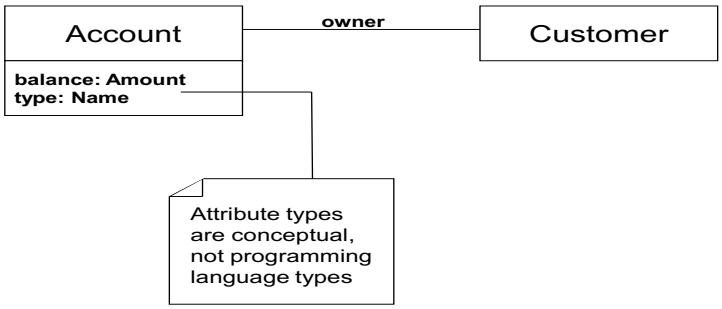
A Partial Domain Class Diagram



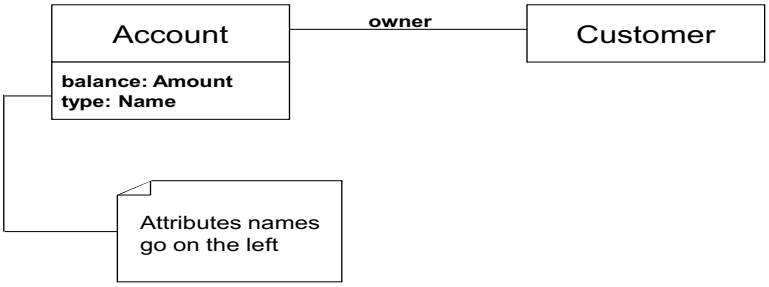
A Partial Domain Class Diagram



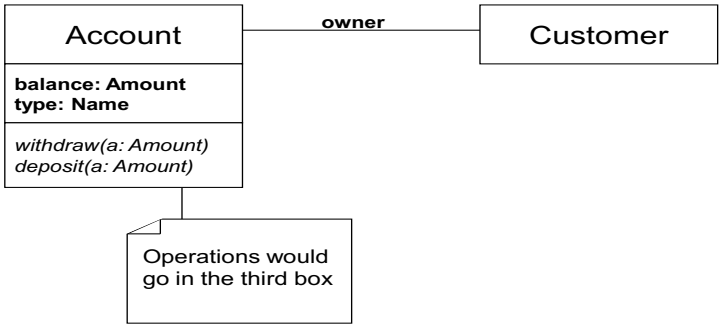
A Partial Domain Class Diagram



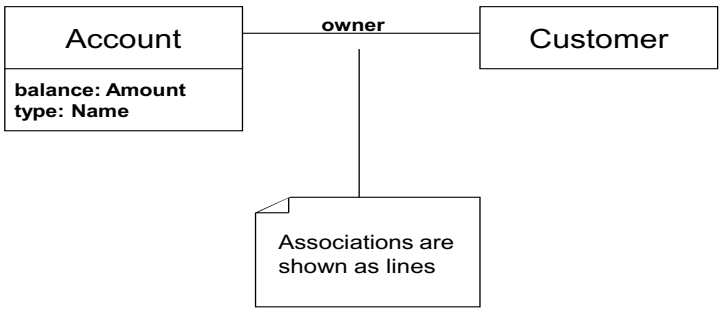
A Partial Domain Class Diagram



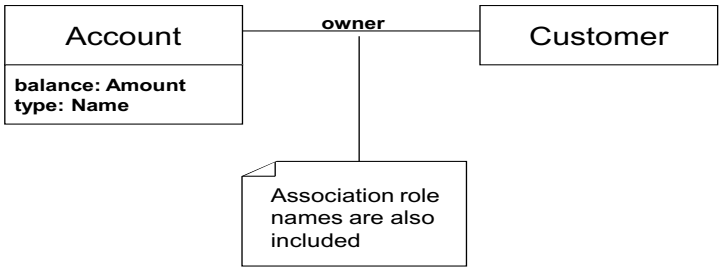
A Partial Domain Class Diagram



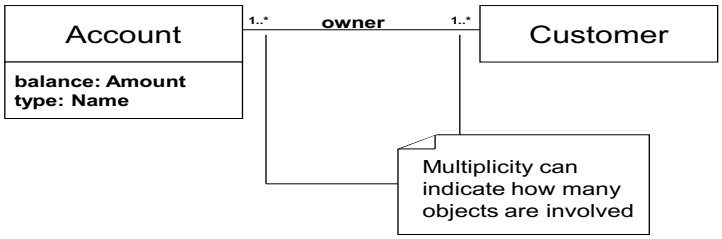
A Partial Domain Class Diagram



A Partial Domain Class Diagram



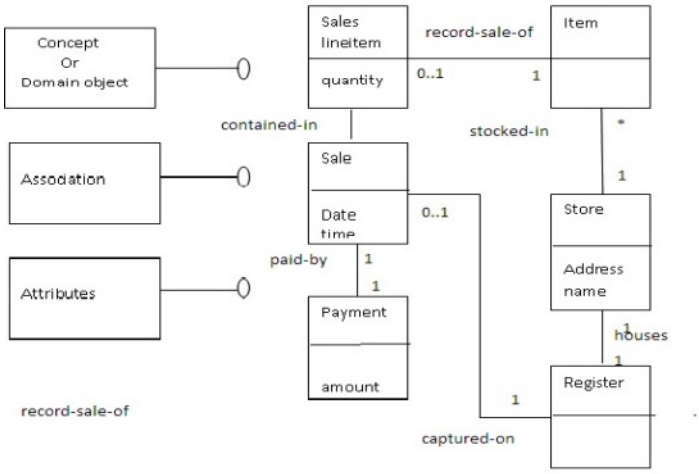
A Partial Domain Class Diagram



EXAMPLE: Partial Domain Model

18

Partial domain model



- Partial domain model drawn with UML class diagram.
- Conceptual classes – payment & sale
- Payment is related to sale
- Sale has date & time attributes

Conceptual classes

- Idea , thing or object
- A conceptual class may be considered in terms of its symbol, intension and extension.

Conceptual classes

- Symbol->word or images representing a conceptual class
- Intension-> The definition of a conceptual class
- Extension-> the set of examples to which the conceptual class applies

Are domain and data model the same thing?

- A domain model is not a data model (which by definition shows persistent data to be stored some where).
- Applying UML notation domain model is illustrated with the set of class diagrams with no operation.
- It shows – domain objects or conceptual classes , associations between classes and attributes of conceptual classes.

How to create a domain model

- Find the conceptual classes
- Draw them as classes in a UML class diagram
- Add associations and attributes

How to find conceptual classes?

Three stages to find conceptual classes

1. Reuse or modify existing models
2. Use a category list
3. Identify noun phrase.

- All 3 strategies should be used initially :
  - even if that leads to much overlapping;
  - it should not take too long anyway;
  - best way to arrive at a rich set of conceptual classes;

Using a Category List

- Use a list of categories and see if they apply within your problem domain : this yields candidate conceptual classes.

Conceptual Class Category	**Examples**
business transactions Guideline: These are critical (they involve money), so start with transactions.	Sale, Payment Reservation
transaction line items Guideline: Transactions often come with related line items, so consider these next.	SalesLineItem
product or service related to a transaction or transaction line item Guideline: Transactions are for something (a product or service).	Item Flight, Seat, Meal

where is the transaction recorded? Guideline: Important.	Register, Ledger
roles of people or organizations related to the transaction; actors in the use case Guideline: We usually need to know about the parties involved in a transaction.	Cashier, Customer, Store MonopolyPlayer Passenger, Airline
place of transaction; place of service	Store Airport, Plane, Seat
noteworthy events, often with a time or place we need to remember	Sale, Payment MonopolyGame Flight
physical objects Guideline: This is especially relevant when creating device-control software, or simulations.	Item, Register Board, Piece, Die Airplane
descriptions of things	ProductDescription FlightDescription
catalogs Guideline: Descriptions are often in a catalog.	ProductCatalog FlightCatalog
containers of things (physical or information)	Store, Bin Board Airplane
things in a container	Item Square (in a Board) Passenger
other collaborating systems	CreditAuthorizationSystem AirTrafficControl

Use a conceptual class category list.

- The creation of a domain model by making a list of candidate conceptual classes. Make a list of candidate concepts

Use noun phrase identification

- Another useful technique(because of this simplicity) suggested in linguistic analysis.
- Identify noun (and noun phrases) in textual descriptions of the problem domain, and consider them as concepts or attributes

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at **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**. Price calculated from a set of **price rules**.

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.
8. System logs completed sale and sends sale and payment information to the external **Accounting** system (for accounting and **commissions**) and **Inventory** system (to update inventory).
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).

- **Extensions (or Alternative Flows): […]**
- 7a. Paying by cash:
- Cashier enters the cash **amount tendered**.
- System presents the **balance due**, and releases the **cash drawer**.
- Cashier deposits cash tendered and returns balance in cash to Customer.
- System records the cash payment

Find and draw conceptual classes

- From the category list and noun phrase analysis, a list is generated of candidate conceptual classes for the domain.
- Immediately draw a UML class diagram of the conceptual classes.

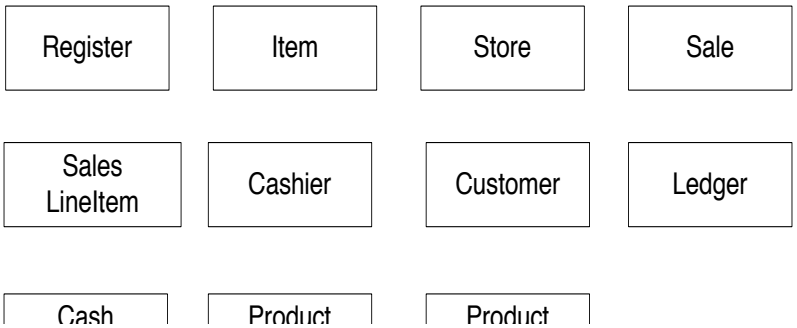
Scenario of process sale

- Sale cashier
- Cash payment customer
- salesLineitem store
- Item ProductDescription
- Register Productcatalog
- Ledger

- Using these approaches we end up with candidate conceptual classes:
  - Some will be outside the current requirements (e.g. price rules);
  - Some will be redundant (e.g. goods is better described by item);
  - Some will be attributes of concepts rather than concepts themselves (e.g. price);

POS Conceptual Classes

- There is no *correct* list of conceptual classes!



- We can also model un-real world problems: E.g. in a telecommunication domain: Message, Connection, Port, Dialog, Route, Protocol would be candidate conceptual classes.
- In the POS application should *Receipt* be a conceptual class?
  - A receipt is certainly noteworthy (allows refunds) : FOR;
  - All the information (? Sure ? E.g. receipt number; or should that be Sale number) on the receipt is however derived from other conceptual classes (from *Sale*) : AGAINST;
  - Returns are not being considering in this iteration : AGAINST;

So we won't include it now: it may be needed when considering the *Handle Returns* use case (Make a note to that effect)
Use terminology (e.g. names of concepts, descriptions of use cases) that make sense in the application context

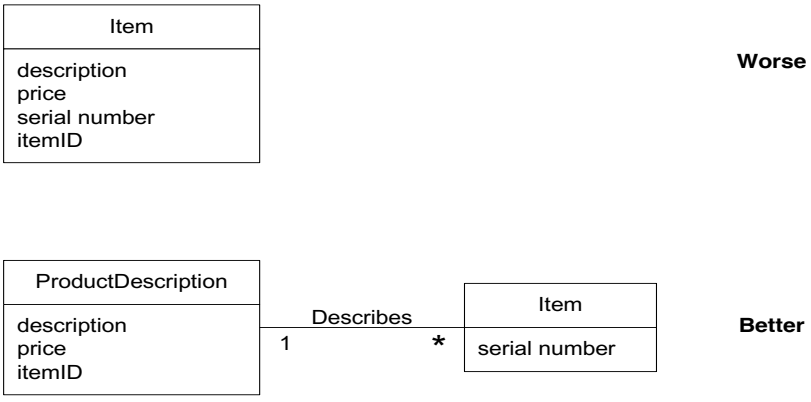
When to model with description classes?

- It contains information that describes something else
- For example a product description that records the price picture and text description of an item

Why use description classes

- The information about the product item may be deleted because of same problem.
- To solve the problem what is need is a product description class that records information about item.
- Product description does not represent an item, it represent a description of information about item.

Why use description classes

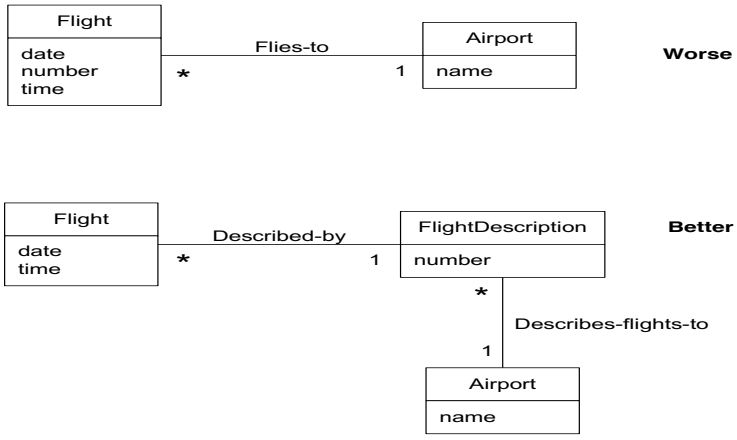


When are description classes useful?
Add a description class (for example, ProductDescription) when:

- There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.

- Deleting instances of things they describe (for example, Item) results in a loss of information that needs to be maintained, but was incorrectly associated with the deleted thing.
- It reduces redundant or duplicated information.

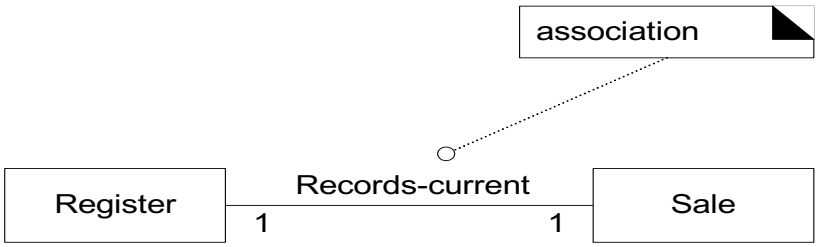
Description in the airline domain



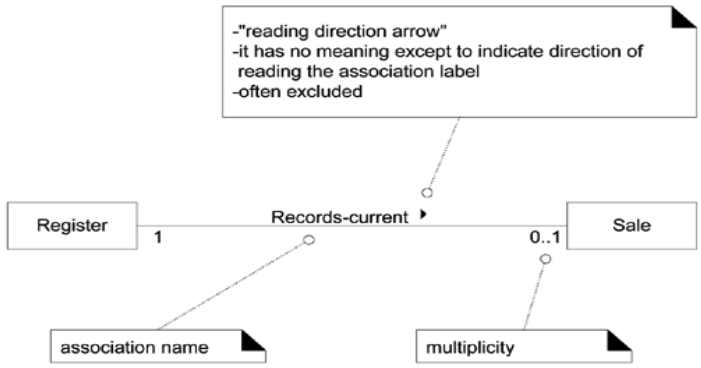
Associations

- An association is a relationship between classes that indicate some meaningful and interesting relationship:
- We must discover all associations the application needs to preserve for some duration (even if it is only for a few milliseconds) and discard all other, theoretical, associations (that simply do not make sense) in our domain model.
- For example, we do need to remember what *SalesLineItem* instances are associated with a *Sale*. For the monopoly domain we need to remember what *Square* a *Piece* is on.

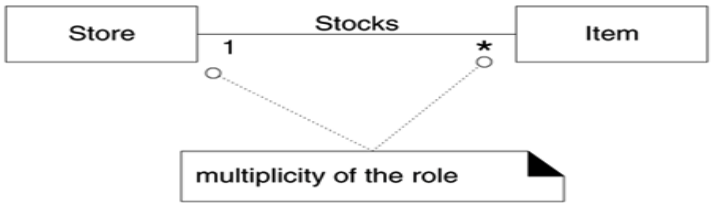
Associations



- An association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible.
- An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation (if absent, the default reading directions are from left to right or top to bottom).

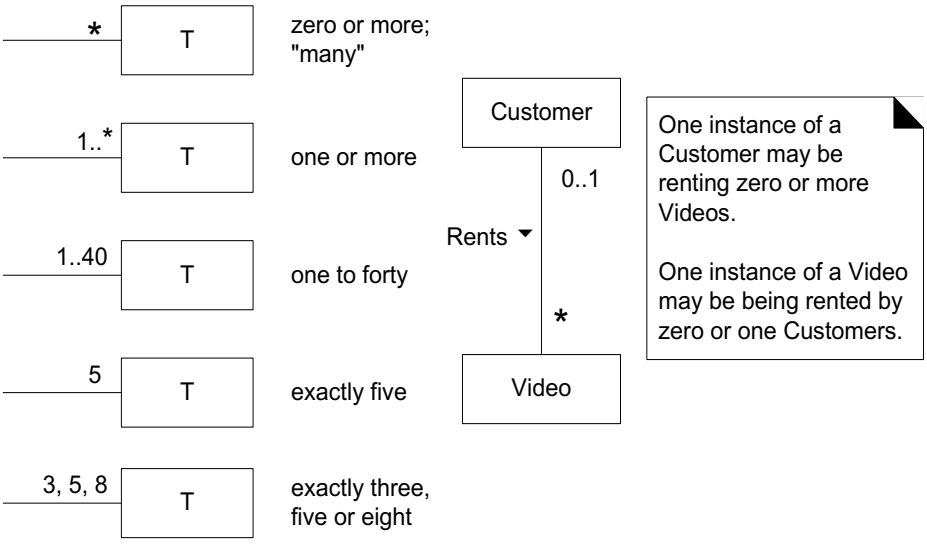


- Properly naming associations is important to enhance understanding: name an association based on a ClassName-VerbPhrase-ClassName format where the verb phrase creates a sequence that is readable and meaningful. (e.g. '*Sale Paid-by CashPayment*')
- Each end of an association is called a role. Roles may optionally have:
  - multiplicity expression
  - name (to clarify meaning)
  - navigability (not relevant during OOA)
- Multiplicity defines how many instances of a class A can be associated with one instance of a class B:



- The multiplicity value communicates how many instances can be validly associated with another, at a particular moment, rather than over a span of time. For example, it is possible that a used car could be repeatedly sold back to used car dealers over time. But at any particular moment, the *Car* is only *Stocked-by* one *Dealer*. No *Car* can be *Stocked-by* many *Dealers* at any particular moment.
- Adding multiplicity values to the roles of an association helps exploring the problem domain.
- Two classes may have multiple associations between them in a UML class diagram; this is not uncommon:

UML: Multiplicity



7

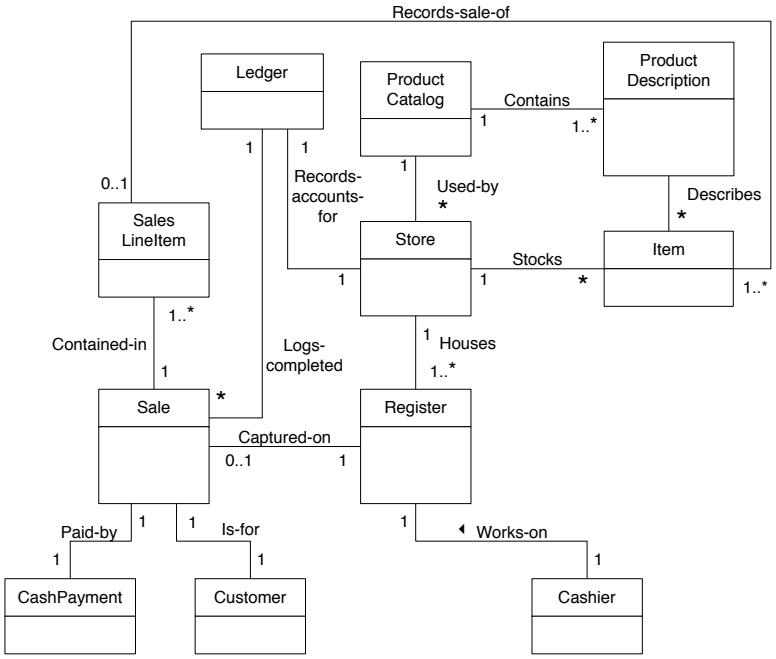
How to Find Associations?

- Two main ways:
  - By reading the current, relevant, requirements and asking ourselves what information is needed to fulfil these requirements: what need to know associations are necessary given our current list of candidate concepts?
  - Using a list of association categories.

Association Category	Examples
A is a transaction related to another transaction B	CashPayment-Sale Cancellation-Reservation
A is a line item of a transaction B	SalesLineItem-Sale
A is a product or service for a transaction (or line item) B	Item-SalesLineItem
A is a role related to a transaction B	Customer-Payment

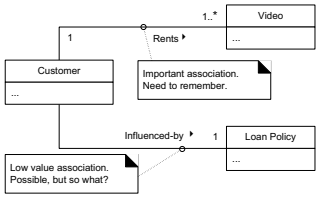
A is a physical or logical part of B	Drawer-Register Square-Board Seat-Airplane

A is physically or logically contained in/on B	Register-Store Item-Shelf
A is a description for B	ProductDescription-Item
A is known/logged/recorded/reported/captured in B	Sale-Register
A is a member of B	Customer-Payment
A is an organizational subunit of B	Cashier-Store
A is an organizational subunit of B	DepartmentStore
Association Category	Examples
A uses or manages or owns B	Cashier-Register Player-Piece
A is next to B	SalesLineItem-SalesLineItem



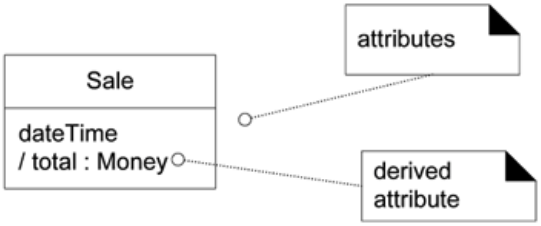
GUIDELINES: Associations

- Only add associations for *noteworthy* relationships. Literally, those for which making a "note" is worthy or business motivated.

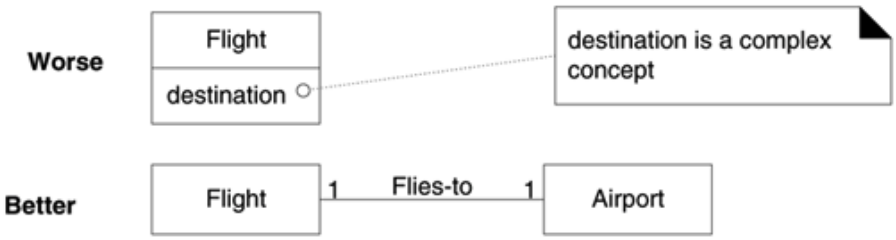


Attributes

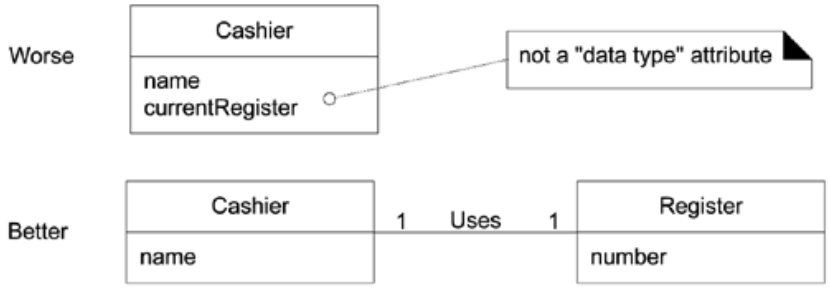
- Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information.
- For example, a receipt (which reports the information of a sale) in the Process Sale use case normally includes a date and time, the store name and address, and the cashier ID, among many other things. Therefore,
  - Sale needs a dateTime attribute.
  - Store needs a name and address.
  - Cashier needs an ID.
- Attributes type and other information may optionally be shown.



- Derived Attributes : The total attribute in the Sale can be calculated or derived from the information in the SalesLineItems. When we want to communicate that 1) this is a noteworthy attribute, but 2) it is derivable, we use the UML convention: a / symbol before the attribute name.
- Primitive data types : attribute types should be what are often thought of as "primitive" data types, such as numbers and booleans. The type of an attribute should not normally be a complex domain concept, such as a Sale or Airport. Nor should they be list of objects or other attributes.
- A common mistake is to model complex information as an attribute when it should be a conceptual class:



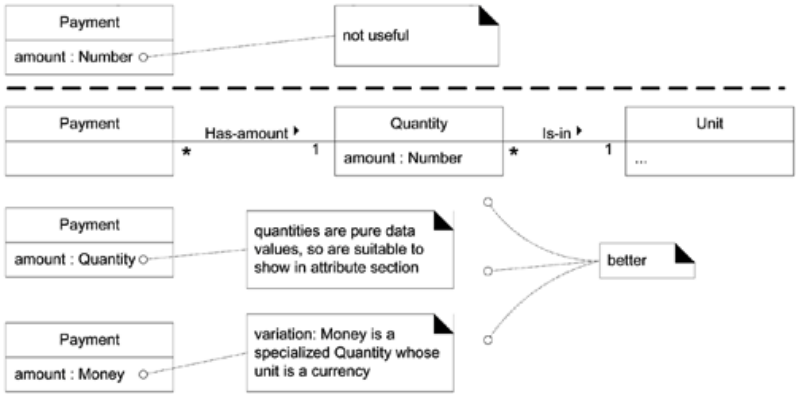
- Another mistake is to relate conceptual classes with an attribute rather than with an association (“foreign key” syndrome):



- Sometimes what appear as a simple data value (using a primitive type such as a string) is actually more complex:
  - We should then create Data Type Classes;
  - For example, a Unique Product Code is not just a number : it has subparts (e.g. contains a manufacturer digit), or can have operations performed on it (e.g. a checksum for validation);
  - An address has subparts.

- Guideline : when should we create new Data Type Classes? Represent what may initially be considered a number or string as a new data type class in the domain model if:
  - It is composed of separate sections.
    - phone number, name of person
  - There are operations associated with it, such as parsing or validation.
    - social security number
  - It has other attributes.
    - promotional price could have a start (effective) date and end date
  - It is a quantity with a unit.
    - payment amount has a unit of currency
  - It is an abstraction of one or more types with some of these qualities.
    - item identifier in the sales domain is a generalization of types such as Universal Product Code (UPC) and European Article Number (EAN)

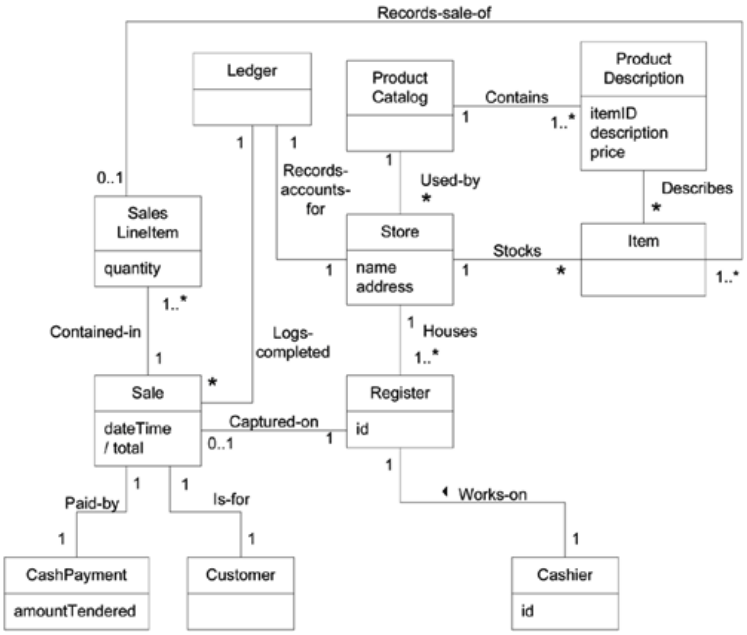
  Examples within the POS case study: See Figure 9.9



Modelling Quantities

Domain Models with Attributes

- For the POS most of the attributes can be obtained by analysing the use case under consideration



POS Partial Domain Model

Conclusion

- When in doubt if the concept is required,keep the concept.
- When in doubt if the the association is required,drop it.
- Do not keep derivable association.

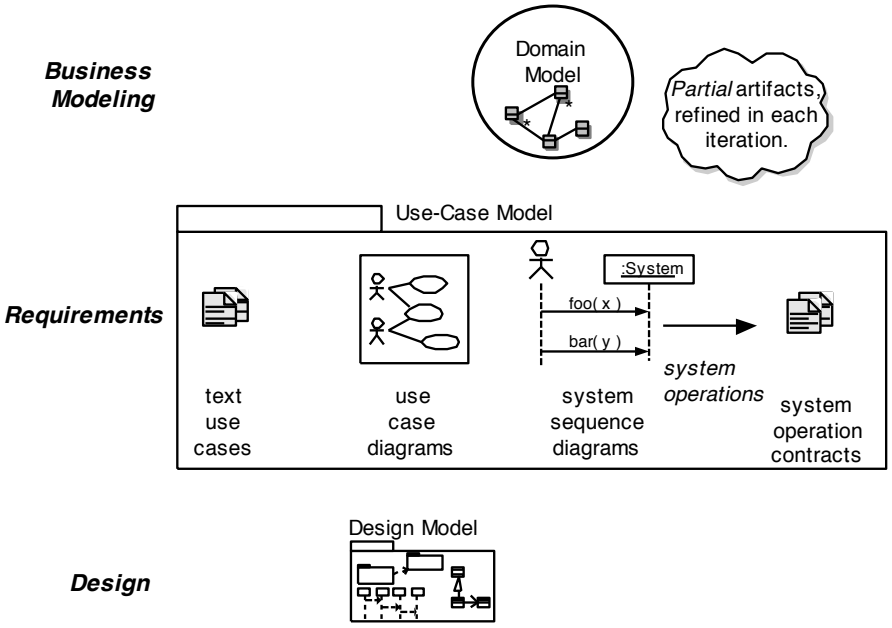
Process: Iterative and Evolutionary Domain Modelling

- In iterative development, we incrementally evolve a domain model over several iterations.
- In each, the domain model is limited to the prior and current scenarios under consideration an expanding to a "big bang" waterfall-style model that early on attempts to capture all possible conceptual classes and relationships.
- For example, this POS iteration is limited to a simplified cash-only Process Sale scenario; therefore, a partial domain model will be created to reflect just that not more.

Domain models within the UP

Discipline	Artifact	Inception	Elaboration	Construction	Transition
Business modeling	Domain model		S		
Requirements	Use case model (SSD)	S	R		
	vision	S	R		
	Supplementary specification	S	R		
	glossary	S	R		
Design	Design model		S	R	
	SW Arch Doc		S		
	Data model		S	r	

Artifacts in the UP Use-Case Model



Domain Model Conclusion

- A relatively useful model has been created for the domain of the POS application.
- A good domain model captures the essential abstractions and information required to understand the domain in context of current requirements, and aids people in understanding the domain – its concepts , terminology, and the relationships.

Refining the Domain Model

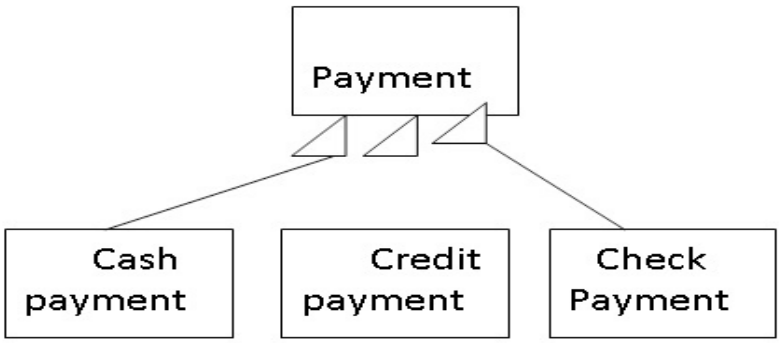
Objectives

- Refine the domain model with generalization, specialization, composition and packages.
- Add association classes to the Domain Model.
- Add aggregation relationships.
- Model the time intervals of applicable information.
- Choose how to model roles.
- Organize the Domain Model into packages.

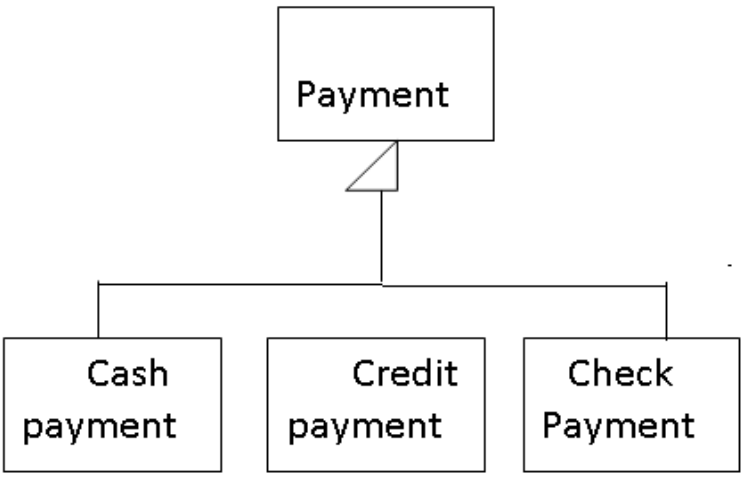
Generalization

- It is the activity of identifying commonality among concepts and defining super class (general concept) and sub class (specialized class) relationships.
- It is a way to construct taxonomy among concepts.

Class hierarchy with separate arrow notation

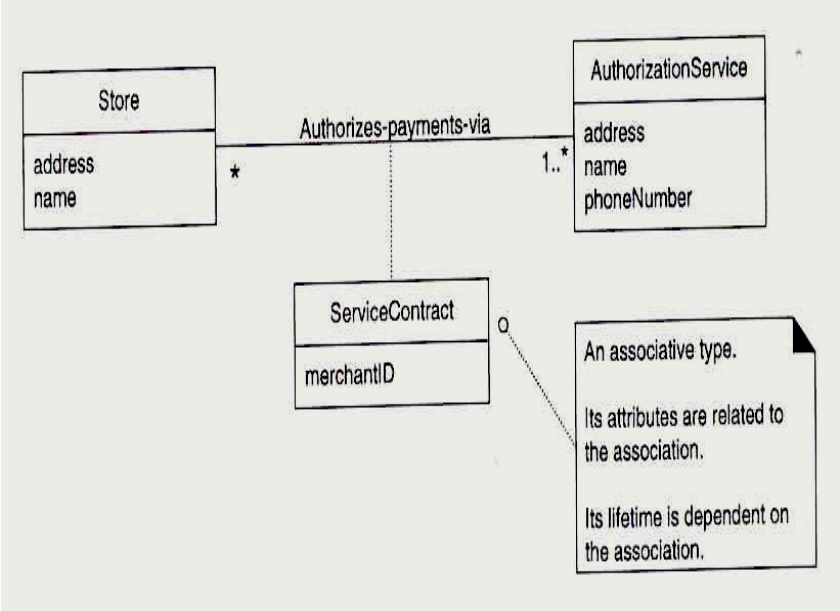


Class hierarchy with shared arrow notation



Class hierarchy - Association Classes

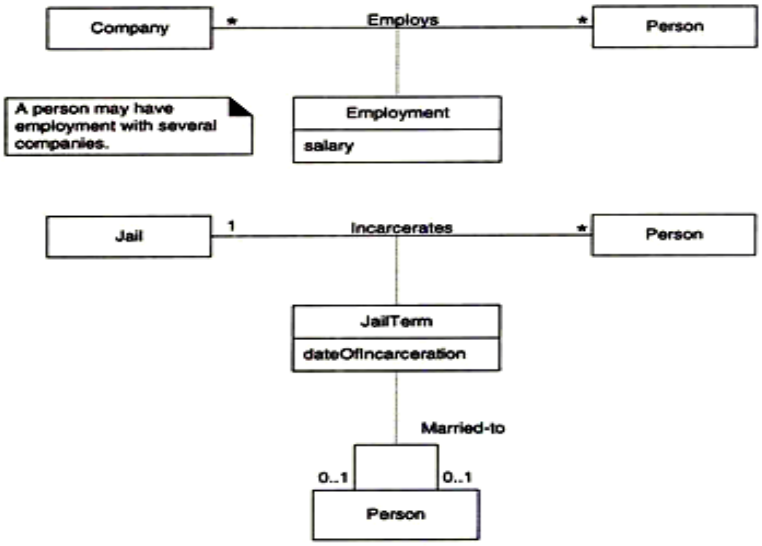
- The notion of an **association class**, in which we can add features to the association itself.
- *ServiceContract* may be modeled as an association class related to the association between *Store* and *AuthorizationSerice*.
- In the UML, this is illustrated with a dashed line from the association to the association class.
- This figure visually communicates the idea that a *ServiceContract* and its attributes are related to the association between *Store* and *AuthorizationService,* and that the lifetime of the *ServiceContract* is dependent on the relationship.



Guidelines For Adding Association Classes

- An attribute is related to an association.
- Instances of the association class have a life-time dependency on the association.
- There is a many-to-many association between two concepts, and information associated with the association itself.

Examples Of Association Classes



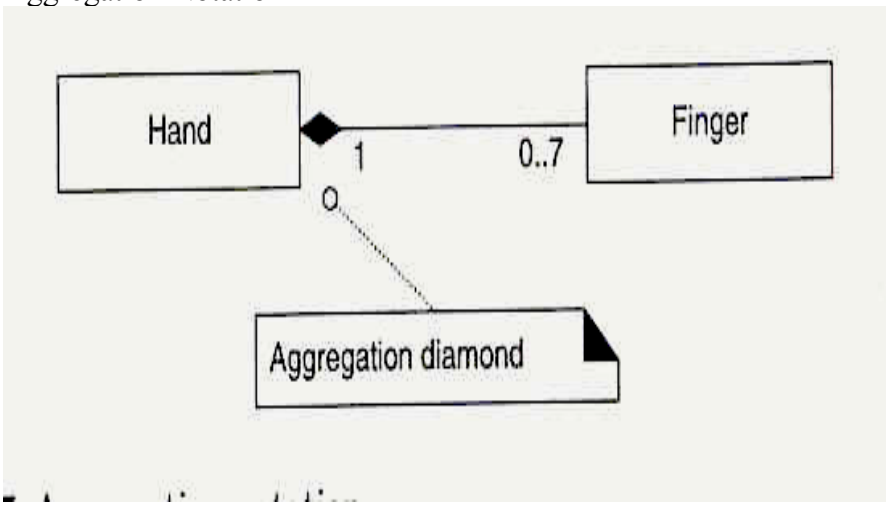
Aggregation and Composition

- **Aggregation** is a kind of association used to model whole-part relationships between things.
- The whole is called **Composite**.
- For instance, physical assemblies are organized in aggregation relationships such as a *Hand* aggregates *Fingers*.

Aggregation in the UML

- Aggregation is shown in the UML with a hollow or filled diamond symbol at the composite end of whole-part association.
- Aggregation is a property of an association role.

Aggregation Notation

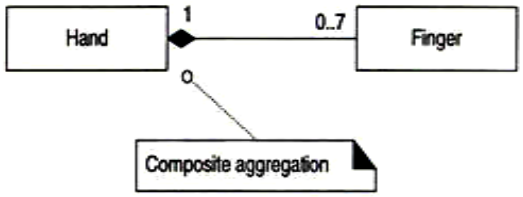


Composite Aggregation – Filled Diamond

- Means that the part is a member of only one composite object, and that there is an existence and disposition dependency of the part on the composite.
- Composition is signified with a filled diamond.
- It implies that the composite solely owns the part, and that they are in a tree structure part hierarchy;it is the most common form of aggregation shown in models.

Example Composition Aggregation

- A finger is a part of at most one hand (we hope!), thus the aggregation diamond is filled to indicate composite aggregation.



Composite Aggregation : In The Design Model

- Composition and its existence dependency implication indicates that composite software objects create the part software objects.
  For example, *Sale* creates *SalesLineItem*.
- It does not represent software objects, the notion of the whole creating the part is seldom relevant.
- For Example, in a "human body" domain model, one thinks of the hand as including the fingers, so if one says, "A hand has come into existence," we understand this to also mean that fingers have come into existence as well.

Multiplicity At Composite End

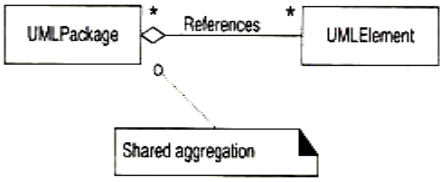
- If the multiplicity at the composite end is exactly one, the part may not exist separate from some composite.
- For Example : if the finger is removed from the hand, it must be immediately attached to another composite object (another hand, a food,…);
- at least, that is what the model is declaring, regardless of the medical merits of this idea!
- If the multiplicity at the composite end is 0…1,then the part may be removed from the composite, and still exist apart from membership in any composite.
- So in previous example if you want fingers floating around by themselves, use 0…1.

Shared Aggregation – Hollow Diamond

- Means that the multiplicity at the composite end may be more than one.
- It is signified with Hollow Diamond.
- Implies that the part may be simultaneously in many composite instances.
- Shared aggregation seldom exists in physical aggregates, but rather in nonphysical concepts.

Shared Aggregation : Example

- A UML package may be considered to aggregate its elements. But an element may be referenced in more than one package.



How To Identify Composition

- In some cases, the presence of Composition is obvious.
  - Usually in physical assemblies.
- But sometimes, it is not clear.
- On Composition: If in doubt, leave it out.

Guidelines For When To Show Composition

- The lifetime of the part is bound within the lifetime of the composite -
  - there is a create – delete dependency of the part on the whole.
- There is an obvious whole-part physical or logical assembly.
- Some properties of the composite propagate to the parts, such as the location.
- Operation applied to the composite propagate to the parts, such as destruction, movement, recording.

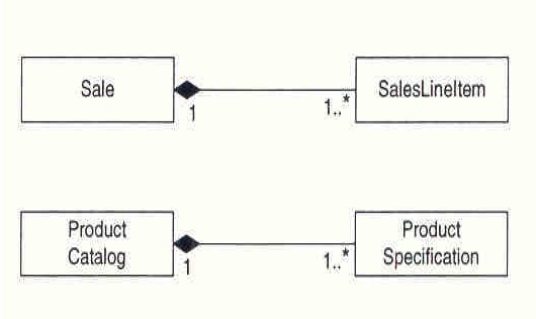
A Benefit Of Showing Composition

- It clarifies the domain constraints regarding the eligible existence of the part independent of the whole. In composite aggregation, the part may not exist outside the lifetime of the whole.
  - During design work, this has an impact on the create – delete dependencies between the whole and part software classes and database elements (in terms of referential integrity and cascading delete paths.
- It assists in the identification of a creator (the composite) using the GRASP Creator pattern.
- Operations – such as copy and delete – applied to the whole often propagate to the parts.

Aggregation In The POS Domain Model

- In the POS domain, the *SalesLineItems* may be considered a part of a composite Sale; in general, transaction line items are viewed as parts of an aggregate transaction.
- In addition to conformance to that pattern, there is create – delete dependency of the line items on the Sale – their lifetime is bound within the lifetime of the sale.
- By similar justification , *Product Catalog* is an aggregate of *Product Specifications.*
- No other relationship is a compelling combination that suggests whole – part semantics, a create – delete dependency, and "If in doubt, leave it out."

Aggregation In POS Application



Time Intervals And Product Prices – Issues "Error

- In the first iteration, SaleLineItems were associated with Product Specifications, that recorded the price of an item.
- This was a reasonable but needs specification.
- It raises the interesting and widely applicable – issue of **time intervals** associated with information, contracts, and so on.

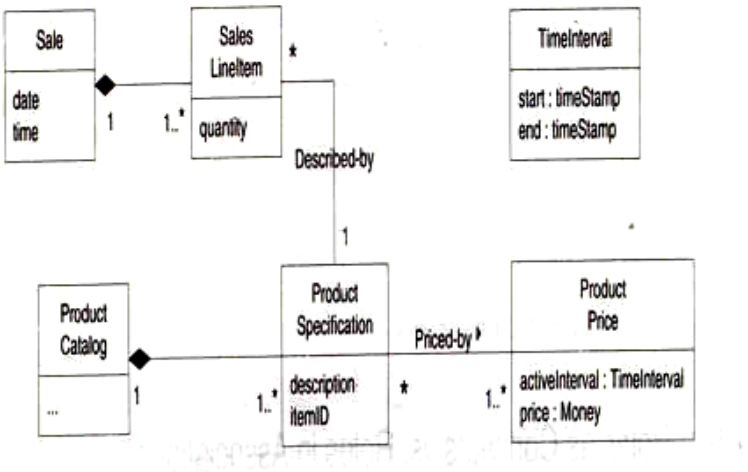
Time Intervals And Product Prices – Issues Examples

- If a *SalesLineItem* always retrieved the current price recorded in Product Specification, then when the price was changed in the object, old sales would refer to new prices, which is incorrect.
- So there is a need for distinction between the historical price when the sale was made, and the current price.

Time Intervals And Product Prices – Fixing The Issues("Error")

- There are Two Ways to solve this issue:
  - One is to simply copy the product price into the *SalesLineItem,* and maintain the current price in the *ProductSpecification.*
  - Another more robust approach, is to associate a collection of *ProductPrices* with a *Product Specification*, each with an associated applicable time interval.
  - Thus, the organization can record all past prices (to resolve the sale price problem, and for trend analysis) and also record future planned prices.

Product Prices And Time Intervals



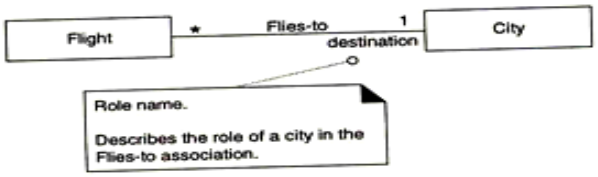
Association Role Names : Properties Overview

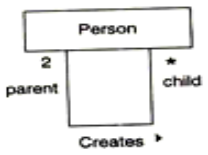
- Each end of association is a role, which has various properties, such as:
  - Name
  - Multiplicity

Association Role Names : Description

- A role name identifies an end of an association and ideally describes the role played by objects in the association.
- An explicit role name is not required – it is useful when the role of the object is not clear.
  - *It usually starts with a lowercase letter.*
- If now explicitly present , assume that the *default role name* is equal to the *related class name*, though starting with a lowercase letter.

Role Names



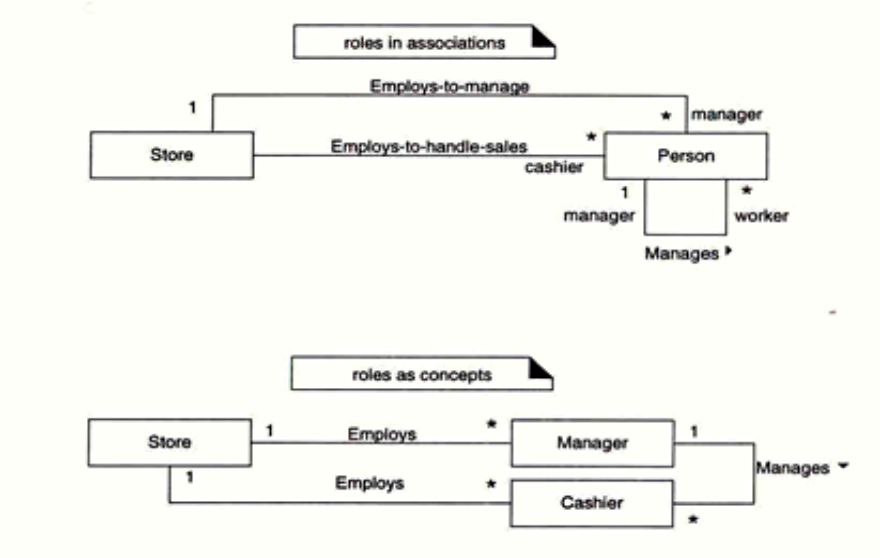


Roles as Concepts *Vs.* Roles In Associations (Overview)

- In a domain model, a real-world role – especially a human role – may be modeled in number of ways such as a discrete concept, or expressed as a role in an association.

- For example, the role of **cashier** and **manager** may be expressed in at least the two ways illustrated in next figure:

Two Ways To Model Human Roles



Roles In Association

- They are a relatively accurate way to express the notation that the same instance of a person takes on multiple (and dynamically changing) roles in various association.
- I , a person, simultaneously or in sequence, may take on the role of writer, object designer, parent, and so on.

Roles As Concepts

- Provides ease and flexibility in adding unique attributes, associations, and additional semantics.
- The implementation of roles as separate classes is easier because of limitations of current popular object – oriented programming languages ,
  - it is not convenient to dynamically mutate an instance of one class into another, or dynamically add behavior and attributes as the role of a person changes.

Derived Elements

- Can be determined from others.
- Attributes and associations are most common derived elements.

Guidelines For Derived Elements

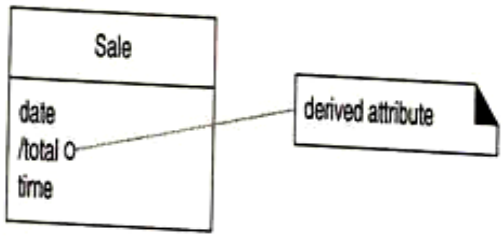
- Avoid showing derived elements in a diagram, since they add complexity without new information.
- However, add a derived element when it is prominent in the terminology, and excluding it impairs comprehension.

Derived Elements – Example

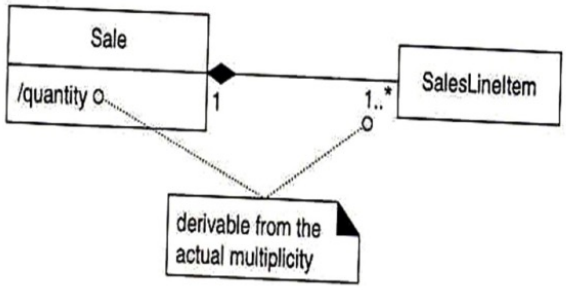
- A Sale *total* can be derived from *SalesLineItem* and *ProductSpecification* information.

- In the UML, it is shown with a "/" preceding the element name.

Derived Attribute



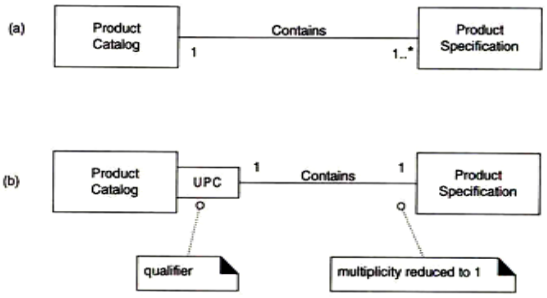
Derived Attribute Related To Multiplicity



Qualified Associations

- A **qualifier** may be used in association;
  it distinguishes the set of objects at the far end of the association based on the qualifier value.
- An association with a qualifier is a **qualified association**.
- For example, *ProductSpecifications* may be distinguished in a *ProductCatalog* by their ItemID.

Qualified Association

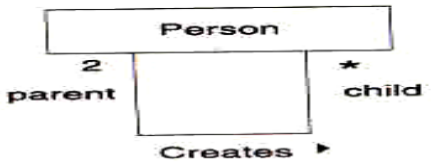


Qualified Association : Explanation

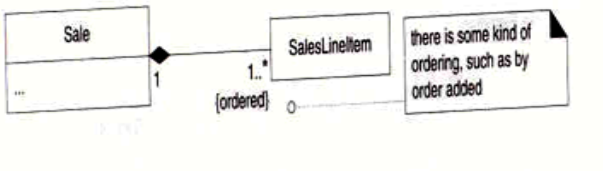
- As contrasted in previous figure (a) vs. (b), qualification reduces the multiplicity at the far end from the qualifier, usually down from many to one.
- Qualifiers do not usually add compelling useful new information, and we can fall into the trap of "design – think".
  However,they can sharpen understanding about the domain

Reflexive Associations

- A concept may have an association to itself;
  this is know as a **reflexive association**.



Ordered Elements



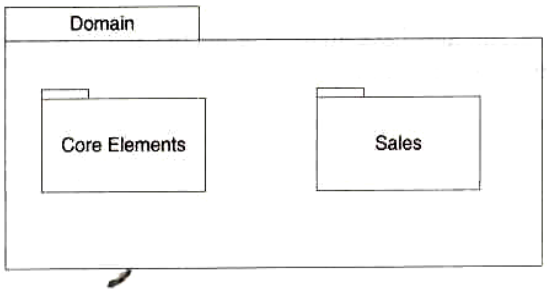
Using Packages To Organize The Domain Model

- A domain model can easily grow large enough that it is desirable to factor it into packages of strongly related concepts,
  - As an aid to comprehension and parallel analysis work in which different people do domain analysis within different sub-domains.
- Following Sections illustrate a package structure for the UP Domain Model :
  - UML Package Notation
  - Ownership and References
  - Package Dependencies

UML Package Notation

- To review, a UML package is shown as a tabbed folder. Subordinate packages may be shown within it.
- The package name is within the tab if the package depicts its elements; otherwise, it is centered within the folder itself.

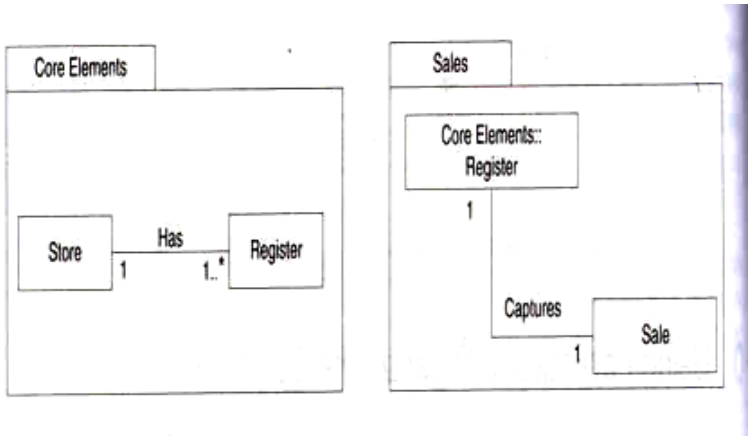
UML Package



Ownership and References

- An element is *owned* by the package within it is defined, but may be *referenced* in other packages.
- In that case, the element name is qualified by the package name using the pathname format *PackageName :: ElementName.*
- A class shown in a foreign package may be modified with new associations, but must otherwise remain unchanged.

A Reference Class In A Package

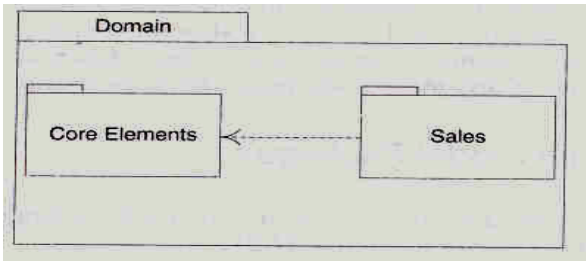


Package Dependencies

- If a model element is in some way dependent on another,
  - The dependency may be shown with a dependency relationship
  - Depicted with Arrowed Line.
- A package dependency indicates
  - elements of the dependent package know about or are coupled to elements in the target package.

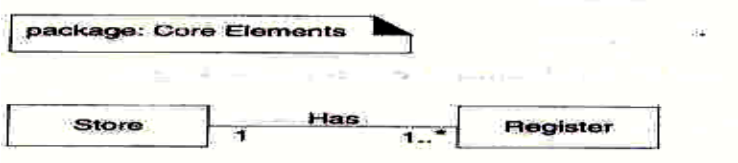
A Package Dependency : Example

- The sales package has a dependency on the Core Elements package.



Package Indication Without Package Diagram

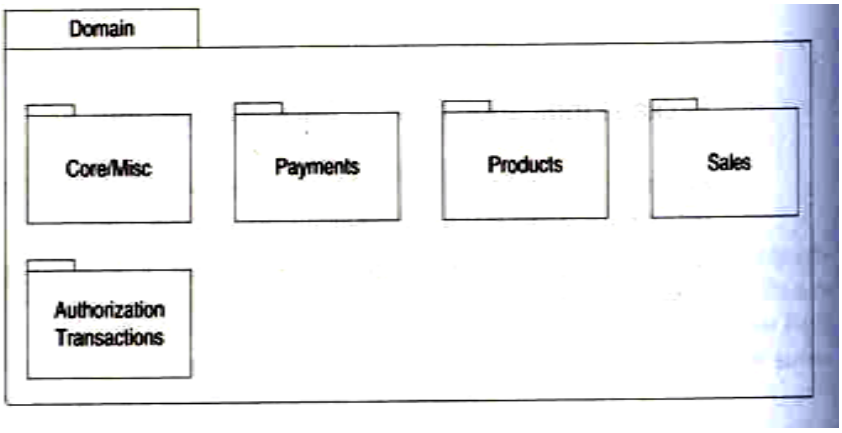
- At times, it is inconvenient to draw a package diagram
  - But it is desirable to indicate the package that the elements are a member of.
  - In this situation, include a note on the diagram



Guidelines On Organizing Classes In Domain Model Within Packages

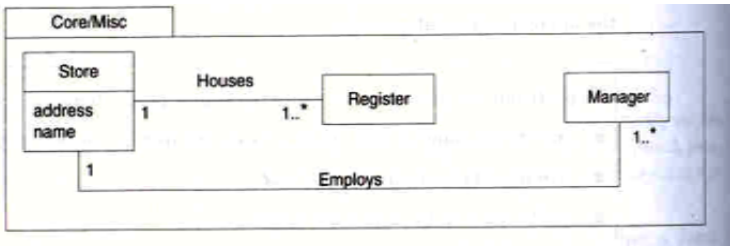
- To partition the domain model into packages, place elements together that:
  - are in the same subject area – closely related by concept or purpose.
  - are in a class hierarchy together.
  - participate in the same use cases.
  - are strongly associated.

POS Domain Model Packages: Domain Concept Packages

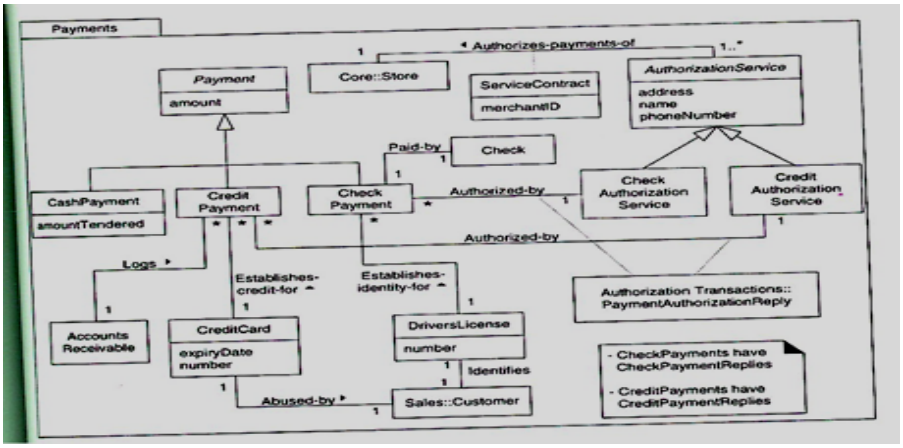


POS Domain Model Packages: Core/Misc Package

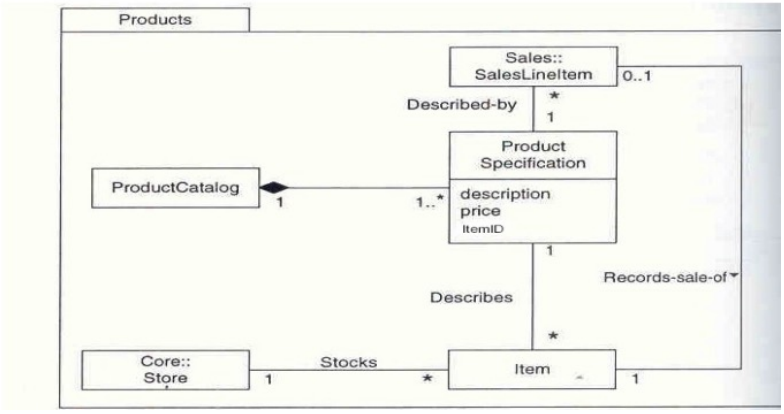
- A Core/Misc package is useful to own widely shared concepts or those without an obvious home.



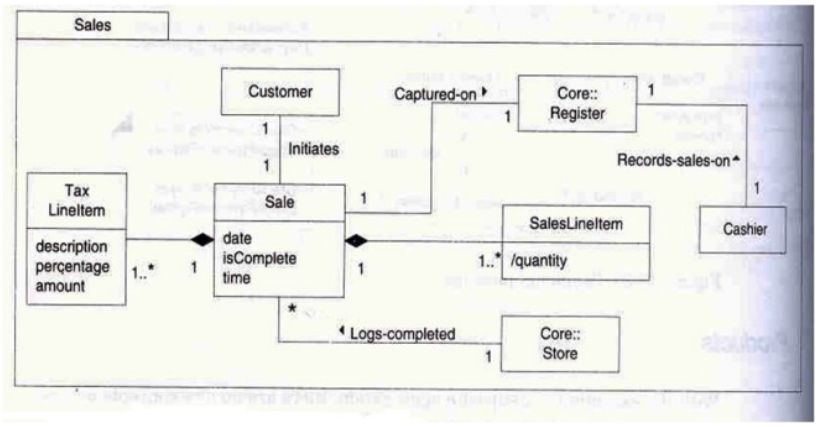
POS Domain Model Packages: Payments Package



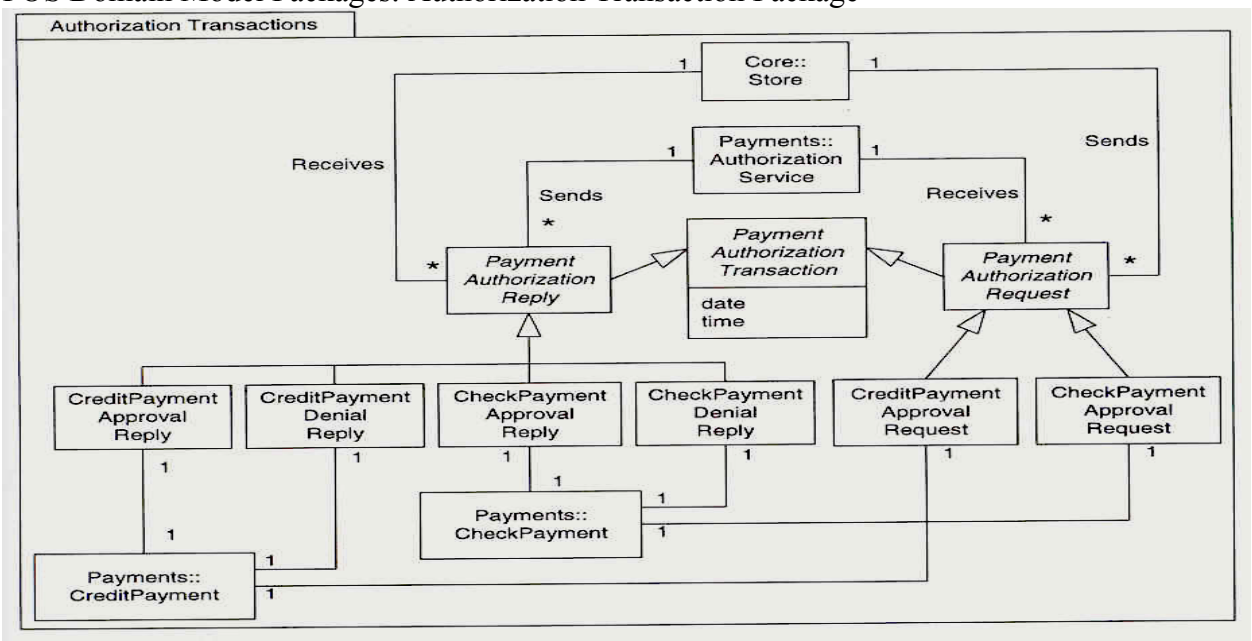
POS Domain Model Packages: Products Package



POS Domain Model Packages: Sales Package



POS Domain Model Packages: Authorization Transaction Package



POS Domain Model Packages:
Authorization Transaction Package