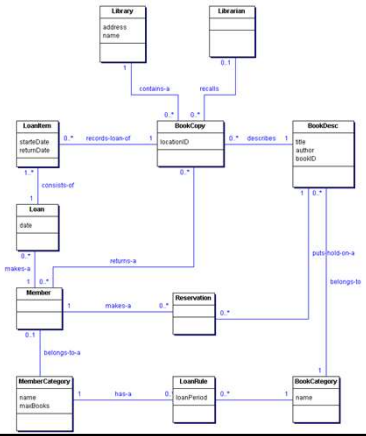
Lecture 04
Domain Models

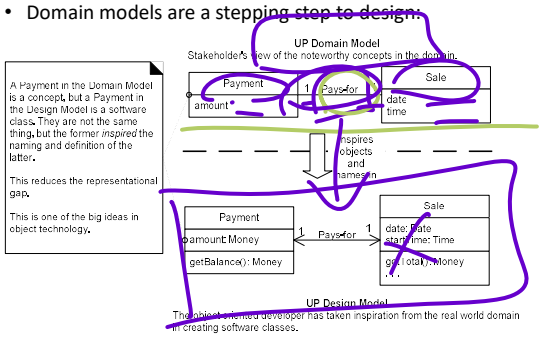
Domain Model

- This is the first OO model that we will see (Use Cases are very useful but are not OO);
- Identifying a rich set of conceptual classes is at the heart of OO analysis;
- A domain model gives a conceptual visualisation of the problem, it shows:
  - domain objects or conceptual classes
  - associations between conceptual classes
  - attributes of conceptual classes
- A domain model does not represent software classes : no methods, programming-related (e.g. interface classes), classes.



Domain Model Example

- Domain models are a stepping step to design.



From Analysis to Design

Introduction (con't)

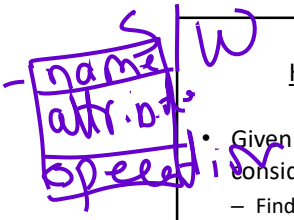
- The investigation of the domain model is bounded (restricted to) the current iteration requirements under consideration.
- Domain model is input to several artifacts; operation contract, design model, objects
- Identify important domain concepts in short time

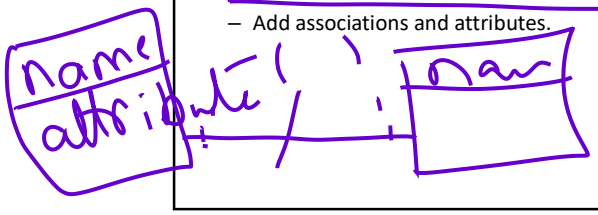
Domain model = Conceptual Model

- Visual representation of real world domain objects
- Why??
- To lower the representation gap
- mental model Vs. software model

How to create a Domain Model?

- Given the current requirements under consideration:
  - Find the conceptual classes.
  - Draw them as classes in a UML class diagram.
  - Add associations and attributes.





How to Find Conceptual Classes?

- 3 main strategies:
  - Reuse or modify existing models! (do this as much as possible!);
  - Think hard and use a category list;
  - Identify noun phrases in requirements;
- 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

Continued …

Conceptual Class Category	Examples
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

Continued …

Conceptual Class Category	Examples
records of finance, work, contracts, legal matters	Receipt, Ledger MaintenanceLog
financial instruments	Cash, Check, LineOfCredit TicketCredit
schedules, manuals, documents that are regularly referred to in order to perform work	DailyPriceChangeList RepairSchedule

Using a Noun Phrases Identification

- Linguistic Analysis – imprecision of natural language is drawback of this technique
- Requirements must be read very closely (especially fully-dressed use cases) and nouns or sequences of nouns identified:
  - This yields candidate conceptual classes.
  - Extracted from
  - Use Case
  - Other documents
  - Domain Experts

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

- 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!

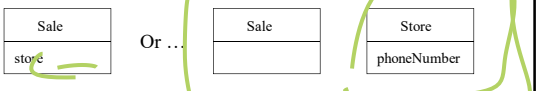
Register	Item	Store	Sale
Sales LineItem	Cashier	Customer	Ledger
Cash Payment	Product Catalog	Product Description	

Initial POS Domain Model

- There is not such thing as correct list of conceptual classes
- Draw – re draw – Ask yourself do you have a good reason to use the updated one??

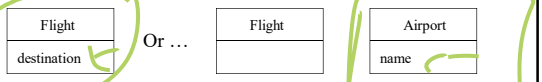
Discussion

- Some conceptual classes will be missed at this stage : aim at a good initial domain, not perfection.
- Common mistake in a domain model; representing something as an attribute when it should be a concept
- Guideline: if something is not a number or a string then it is probably a conceptual class, not an attribute.
  E.g.



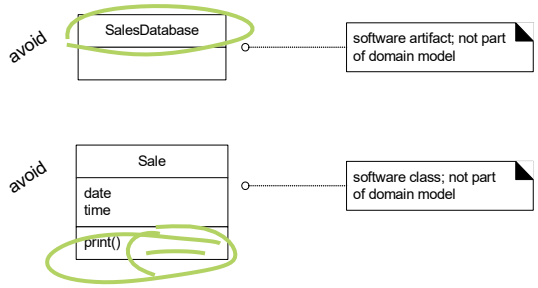
Here, since a store can have many interesting attributes (it is not a simple string) it should be made a separate concept.

- Another example:



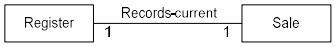
- The converse mistake, representing something as a concept when it should be an attribute, on the other hand is not a problem as it is easily fixed.

- Another common mistake is to include a *database* concept: whether some of the information will be held in a database is a design decision (it is an implementation detail) : it is wrong to include it in a domain model.



Associations

- An association is a relationship between classes that indicate some meaningful relationship:



- We must discover all associations the application needs to preserve for some duration 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*.

- On the other hand, while a Cashier may do a product look-up via a POS terminal, the system has no need to remember the fact that a *Cashier* has looked-up a particular *ProductDescription*:

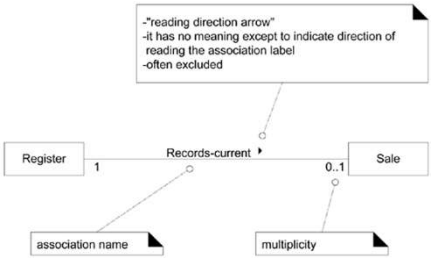
  Important; associations are not there to record actions that actors may perform; they are there to support the information requirements of the processes that need to be implemented.

- We must only focus on the "need to remember" associations to avoid adding too many associations.

- Eventually, the associations that we discover will likely end up being implemented as path of navigability between objects (using pointers to objects) : but we should not worry about this during OOA.

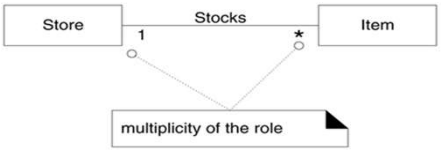
- The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.

- 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).



Associations in UML

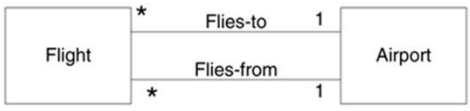
- Properly naming associations is important to enhance understanding: name an association based on a ClassName-VerbPhrase-ClassName format where the verb phrase creates a sequence that is readable and meaningful. (e.g. '*Sale Paid-by CashPayment*')
- Each end of an association is called a role. Roles 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:



Multiplicity Values

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.

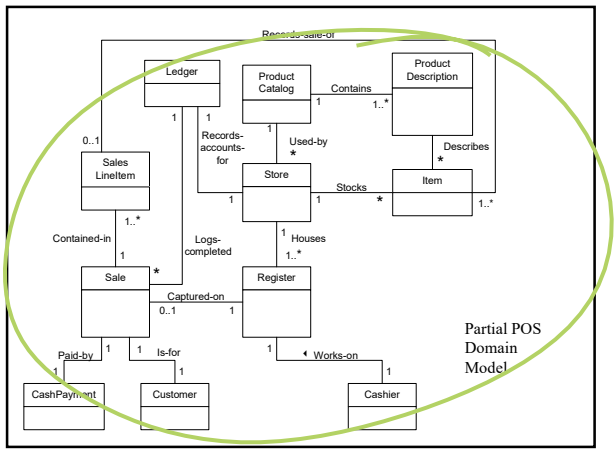
Association Category	Examples
A is a transaction related to another transaction B	CashPayment-Sale Cancellation-Reservation
A is a line item of a transaction B	SalesLineItem-Sale
A is a product or service for a transaction (or line item) B	Item-SalesLineItem Flight-Reservation
A is a role related to a transaction B	Customer-Payment Passenger-Ticket
A is a physical or logical part of B	Drawer-Register Square-Board Seat-Airplane

Continued …

Association Category	Examples
A is physically or logically contained in/on B	Register-Store Item-Shelf Square-Board Passenger-Airplane
A is a description for B	ProductDescription-Item FlightDescription-Flight
A is known/logged/recorded/reported/captured in B	Sale-Register Piece-Square
A is a member of B	Customer-Payment Passenger-Ticket
A is an organizational subunit of B	Cashier-Store Player-MonopolyGame Pilot-Airline
A is an organizational subunit of B	DepartmentStore MaintenanceAirline

Continued …

Association Category	Examples
A uses or manages or owns B	Cashier-Register Player-Piece Pilot-Airplane
A is next to B	SalesLineItem-SalesLineItem Square-Square City-City

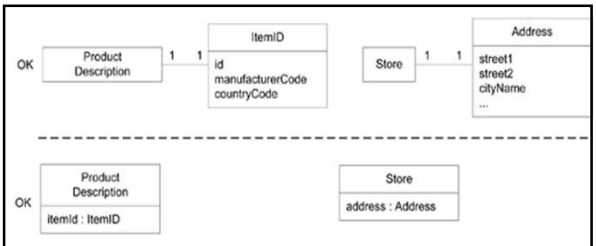


Partial POS Domain Model

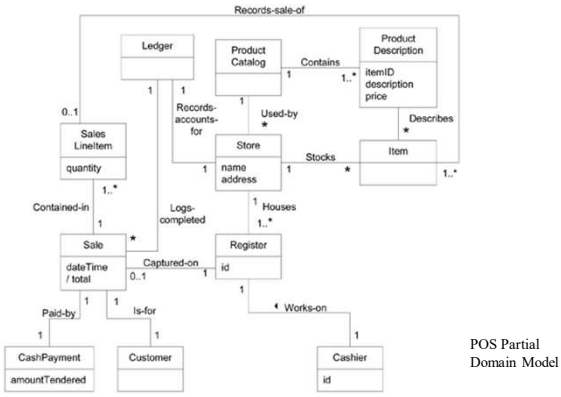
Attributes

- 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.

- 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)



Different Ways to Represent Data Type Classes in the UML



POS Partial Domain Model

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, rather than 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.

Activity: Draw domain model