# Lecture 2

# Object Oriented Analysis

- **OO Analysis Overview**
- **Domain Modeling**

# Understanding Analysis

- **In software engineering, analysis is the process of converting the user requirements to system specification**
  - system means the software to be developed.
- **System specification, also known as the logic structure, is the developer's view of the system.**
- **Function-oriented analysis – concentrating on the decomposition of complex functions to simpler ones.**
- **Object-oriented analysis – identifying objects and the relationship between objects.**

# Understanding Analysis

- **The goal in Analysis is to understand the problem**
  - and to begin to develop a visual model of what you are trying to build, independent of implementation and technology concerns.
- **Analysis focuses on translating the functional requirements into software concepts.**
  - The idea is to get a rough cut at the objects that comprise our system, but focusing on behavior (and therefore encapsulation).
  - We then move very quickly, nearly seamlessly into "design" and the other concerns.

# Object Oriented Analysis

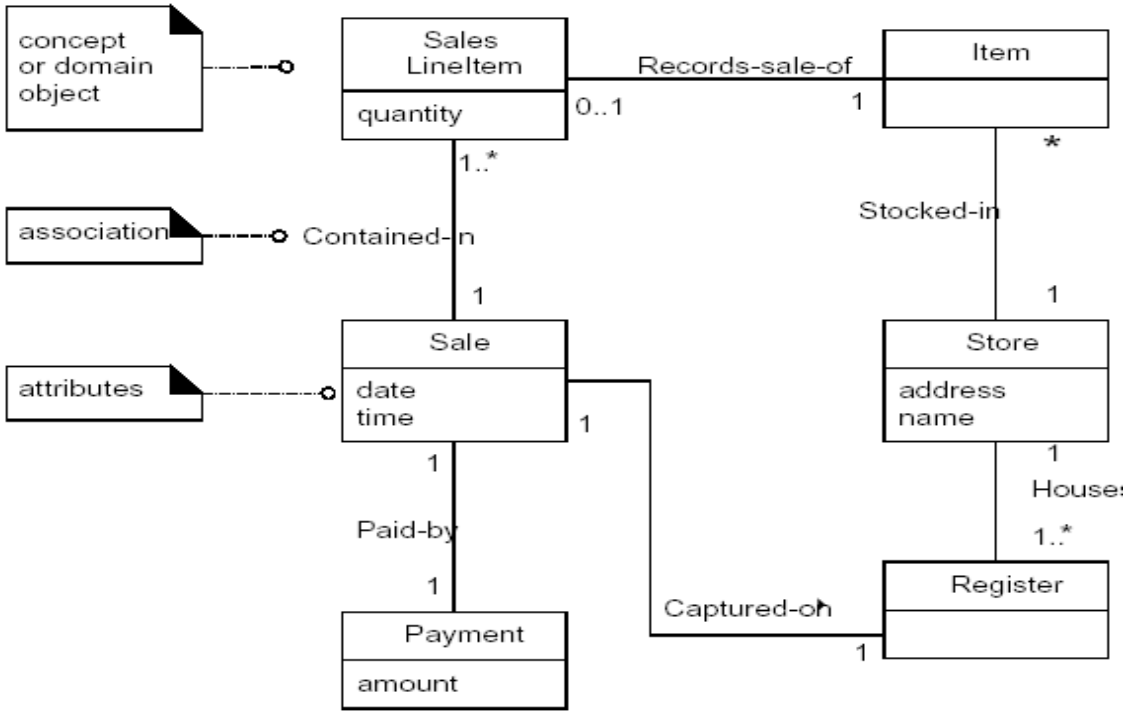
- Identifying objects:
  - Using concepts, CRC cards, stereotypes, etc.
- Organising the objects:
  - classifying the objects identified, so similar objects can later be defined in the same class.
- Identifying relationships between objects:
  - this helps to determine inputs and outputs of an object.
- Defining operations of the objects:
  - the way of processing data within an object.
- Defining objects internally:
  - information held within the objects.

# Domain Models

- **A domain model is a representation of real-world conceptual classes**
  - not a representation of software components.
  - not a set of diagrams describing software classes,
  - not software objects with responsibilities.
- **A domain model** is a visual representation of conceptual classes or real-world objects in a domain of interest [MO95, Fowler96]
  - They have also been called conceptual models, domain object models, and analysis object models.
- The UP defines a Domain Model as one of the artifacts that may be created in the Business Modeling discipline.

# Domain Models

- Using UML notation, a domain model is illustrated with a set of **class diagrams** in which no operations are defined. It may show:
  - domain objects or conceptual classes
  - associations between conceptual classes
  - attributes of conceptual classes

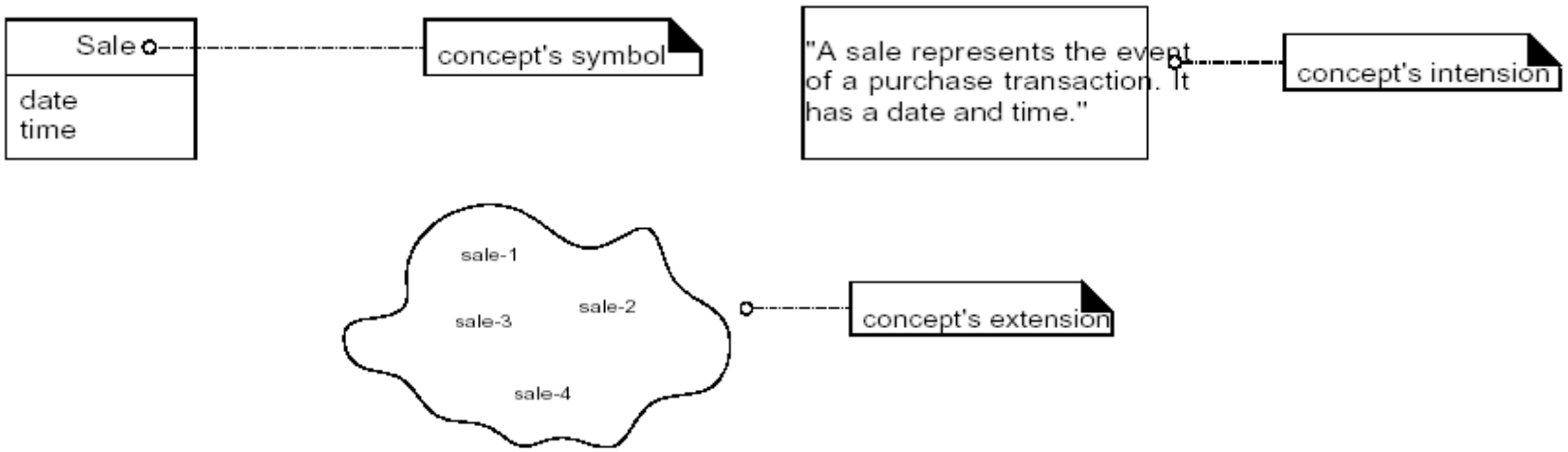


# Domain Models

- The domain model illustrates conceptual classes or vocabulary in the domain.
- Domain Models are not models of software components. The following elements are not suitable in a domain model:
  - Software artifacts, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.
  - Responsibilities or methods.

# Domain Models

- The domain model illustrates conceptual classes or vocabulary in the domain.
- Informally, a conceptual class is an idea, thing, or object.
- More formally, a conceptual class may be considered in terms of its symbol, intension, and extension[MO95].
  - Symbol—words or images representing a conceptual class.
  - Intension—the definition of a conceptual class.
  - Extension—the set of examples to which the conceptual class applies.



# Domain Models

- Software problems can be complex;
  - Decomposition (divide-and-conquer) is a common strategy to deal with this complexity by division of the problem space into comprehensible units.
  - In structured analysis, the dimension of decomposition is by processes or *functions.*
  - However, in object-oriented analysis, the dimension of decomposition is fundamentally by things or entities in the domain.
- A central distinction between object-oriented and structured analysis is: division by conceptual classes (objects) rather than division by functions.
- A primary analysis task is to identify different concepts in the problem domain and document the results in a domain model

# Conceptual Class Identification

- A useful guideline in identifying conceptual classes:
  - It is better to overspecify a domain model with lots of fine-grained conceptual classes than to underspecify it.
- Strategies to Identify Conceptual Classes.
  - Use a conceptual class category list.
  - Finding conceptual classes with Noun Phrase Identification
- Another excellent technique for domain modeling is the use of **analysis patterns,** which are existing partial domain models created by experts,
  - using published resources such as *Analysis Patterns* [Fowler96] and *Data Model Patterns*[Hay96].

# Domain Modeling Guidelines

- *How to Make a Domain Model*
  - 1. List the candidate conceptual classes using the Conceptual Class Category List and noun phrase identification techniques related to the current requirements under consideration.
  - 2. Draw them in a domain model.
  - 3. Add the associations necessary to record relationships for which there is a need to preserve some memory.
  - 4. Add the attributes necessary to fulfill the information requirements.
- *On Naming and Modeling Things*
  - *use the vocabulary of the domain when naming conceptual classes and attributes.*
    - **For example, if developing a model for a library, name the customer a *"Borrower"* or *"Patron"*—the terms used by the library staff.**
  - a domain model may exclude conceptual classes in the problem domain not pertinent to the requirements.
    - **For example, we may exclude *Pen* and *PaperBag* from our domain model (for the current set of requirements) since they do not have any obvious noteworthy role.**
  - the domain model should exclude things *not* in the problem domain under consideration.

# Domain Modeling Guidelines

- *A Common Mistake in Identifying Conceptual Classes*
  - Perhaps the most common mistake when creating a domain model is to represent something as an attribute when it should have been a concept.
  - A rule of thumb to help prevent this mistake is:
    - **If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.**
- As an example, should *store* be an attribute of *Sale,* or a separate conceptual class *Store?*
  - In the real world, a store is not considered a number or text - the term suggests a legal entity, an organization, and something occupies space. Therefore, *Store* should be a concept.
- As another example, consider the domain of airline reservations. Should *destination* be an attribute *of Flight,* or a separate conceptual class *Airport?*
  - In the real world, a destination airport is not considered a number or text—it is a massive thing that occupies space. Therefore, *Airport* should be a concept.

# Resolving Similar Conceptual Classes

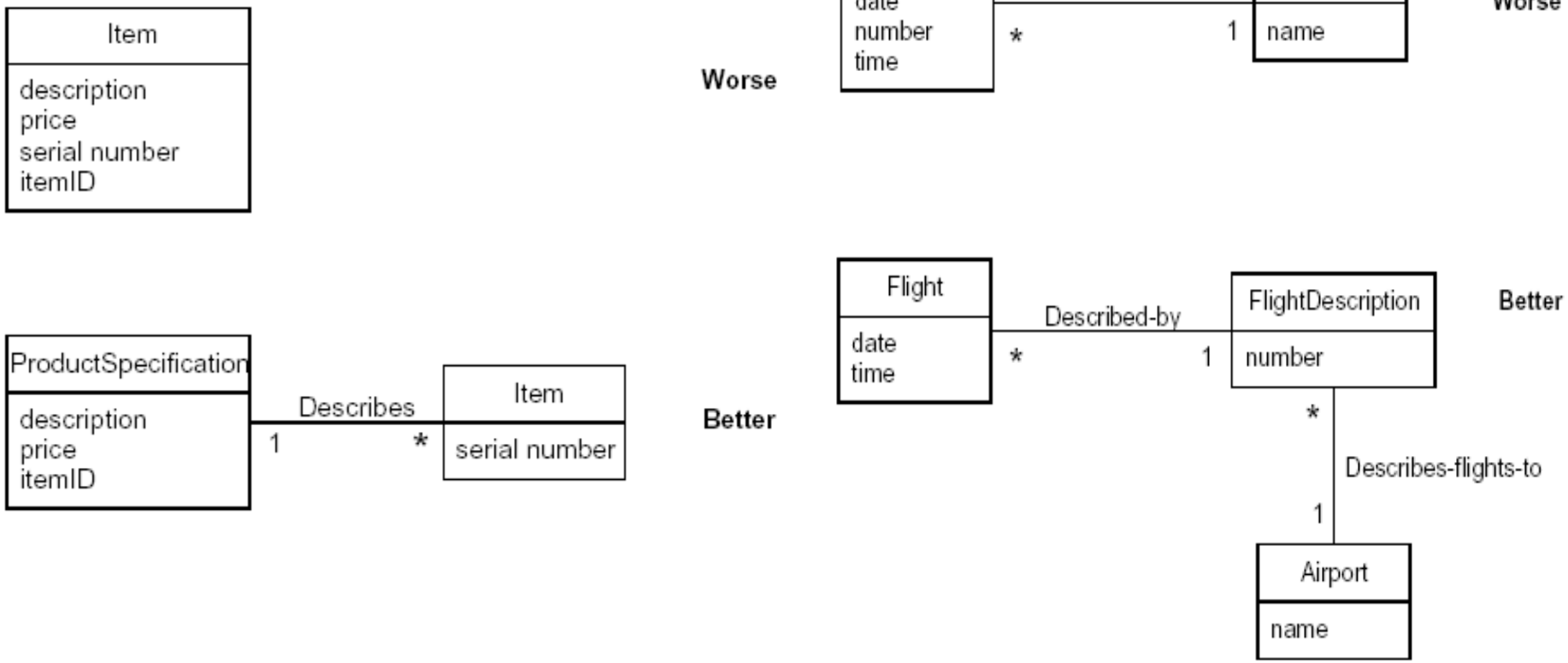
- As an example, in the domain model, should the symbol *Register* be used instead *of POST?*
  - *POST"* is a term familiar in the territory, so it is a useful symbol from the point of view of familiarity and communication.
  - By the goal of creating models that represent abstractions and are implementation independent, *Register* is appealing and useful.
- **As a rule of thumb, a domain model is not absolutely correct or wrong, but more or less useful; it is a tool of communication.**

# Modeling the *Unreal world*

- Some software systems are for domains that find very little analogy in natural or business domains;
  - software for telecommunications is an example.
- It is still possible to create a domain model in these domains, but it requires a high degree of abstraction and stepping back from familiar designs.
- For example, here are some candidate conceptual classes related to a telecommunication switch:
  - *Message, Connection, Port, Dialog, Route, Protocol.*

# Specification or Description Conceptual Classes

- Specifications or descriptions about other things.



# Specification or Description Conceptual Classes

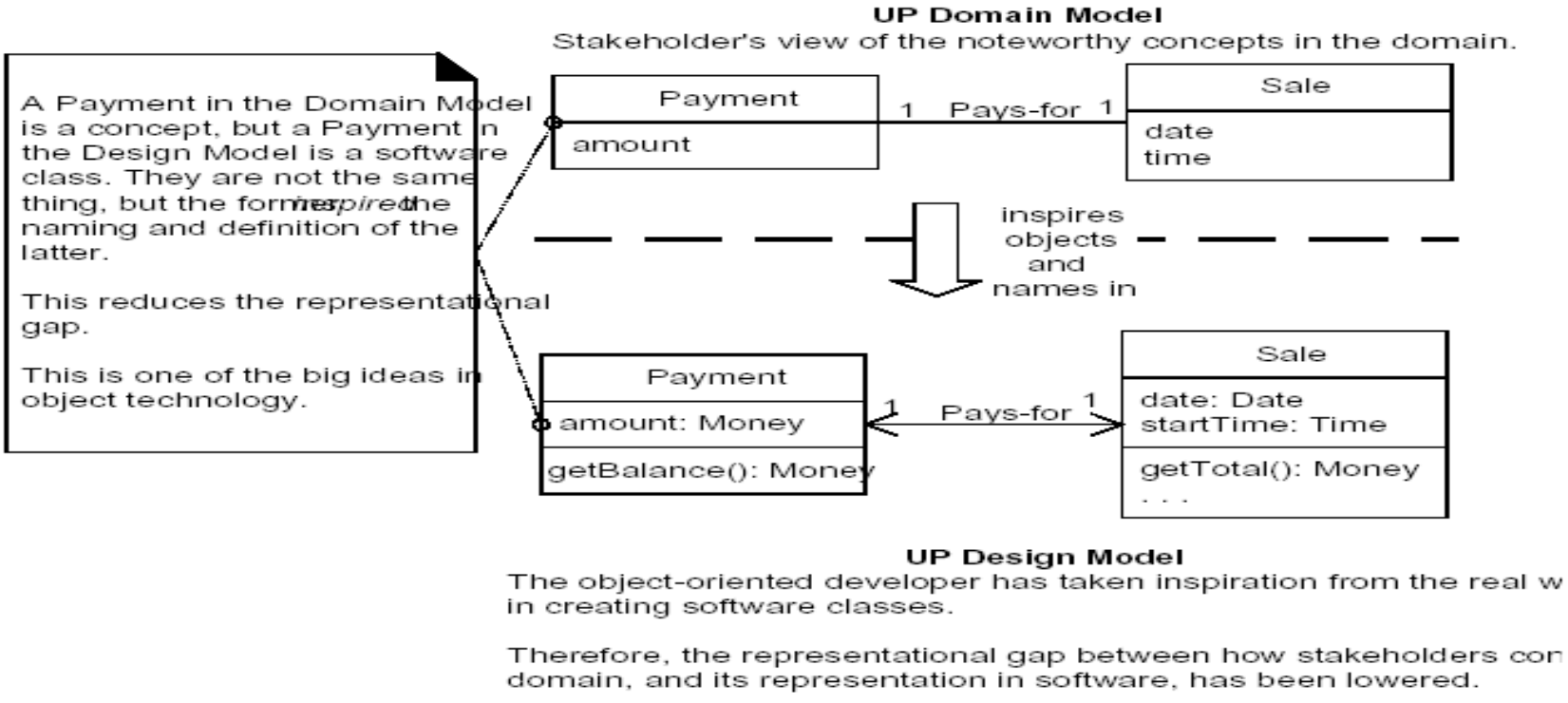
- Add a specification or description conceptual class (for example, *ProductSpecification)* when:
  - There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
  - Deleting instances of things they describe (for example, *Item)* results in a loss of information that needs to be maintained, due to the incorrect association of information with the deleted thing.
  - It reduces redundant or duplicated information.

# UML Notation v.s. Methodology

- The UML simply describes raw diagram types, such as class diagrams and sequence diagrams. It does not superimpose a method or modeling perspective on these.
- Rather, a process (such as the UP) applies raw UML in the context of methodologist-defined models.
- Three perspectives and types of models:
  - Conceptual perspective
    - **the diagrams are interpreted as describing things in the real world or domain of interest.**
  - Specification perspective
    - **the diagrams are interpreted as describing software abstractions or**
    - **components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).**
  - Implementation perspective
    - **the diagrams are interpreted as describing software implementations in a particular technology and language (such as Java).**

# Lowering the Representational Gap

- The Domain Model provides a visual dictionary of the domain vocabulary and concepts from which to draw inspiration for the naming of some things in the software design.
- In object design and programming it is common to create software classes whose names and information is inspired from the real world domain.

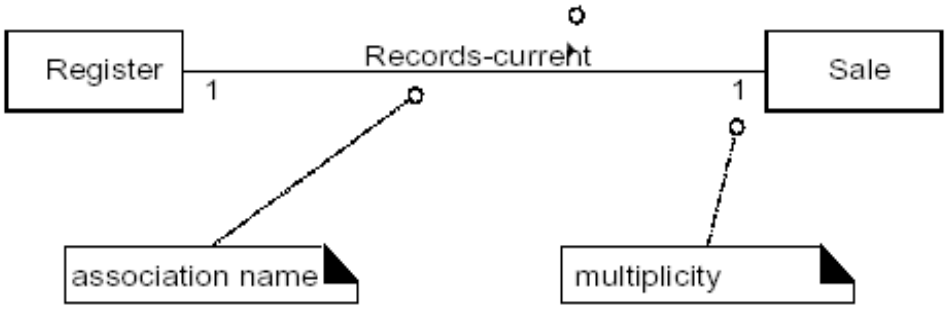


# Associations

- An **association** is a relationship between types (or more specifically, instances of those types) that indicates some meaningful and interesting connection
- In the UML associations are defined as "the semantic relationship between two or more classifiers that involve connections among their instances.“
- *Criteria for Useful Associations*
  - Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
  - Associations derived from the Common Associations List.

# The UML Association Notation

- An association is represented as a line between classes with an association name.
  - The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible.
  - This traversal is purely abstract; it is not a statement about connections between software entities.
  - 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.



# Finding Associations

- Start the addition of associations by using the Common Associations List
  - It contains common categories that are usually worth considering.
- Here are some high-priority association categories that are invariably useful to include in a domain model:
  - A is a *physical or logical part* of B.
  - A is *physically or logically contained* in/on B.
  - A is *recorded in* B.

| Category | Examples |
|---|---|
| A is a physical part of B | *Drawer — Register (or more specifically, a POST)*<br>*Wing — Airplane* |
| A is a logical part of B | *SalesLineItem — Sale*<br>*FlightLeg—FlightRoute* |
| A is physically contained in/on B | *Register — Store, Item — Shelf*<br>*Passenger — Airplane* |
| A is logically contained in B | *ItemDescription — Catalog*<br>*Flight— FlightSchedule* |
| A is a description for B | *ItemDescription — Item*<br>*FlightDescription — Flight* |
| A is a line item of a transaction or report B | *SalesLineItem — Sale*<br>*Maintenance Job — Maintenance-Log* |
| A is known/logged/recorded/reported/captured in B | *Sale — Register*<br>*Reservation — FlightManifest* |
| A is a member of B | *Cashier — Store*<br>*Pilot — Airline* |
| A is an organizational subunit of B | *Department — Store*<br>*Maintenance — Airline* |
| A uses or manages B | *Cashier — Register*<br>*Pilot — Airplane* |
| A communicates with B | *Customer — Cashier*<br>*Reservation Agent — Passenger* |
| A is related to a transaction B | *Customer — Payment*<br>*Passenger — Ticket* |
| A is a transaction related to another transaction B | *Payment — Sale*<br>*Reservation — Cancellation* |
| A is next to B | *SalesLineItem — SalesLineItem*<br>*City— City* |

| Category | Examples |
|---|---|
| A is owned by B | *Register — Store*<br>*Plane — Airline* |
| A is an event related to B | *Sale — Customer, Sale — Store*<br>*Departure — Flight* |

# Association Guidelines

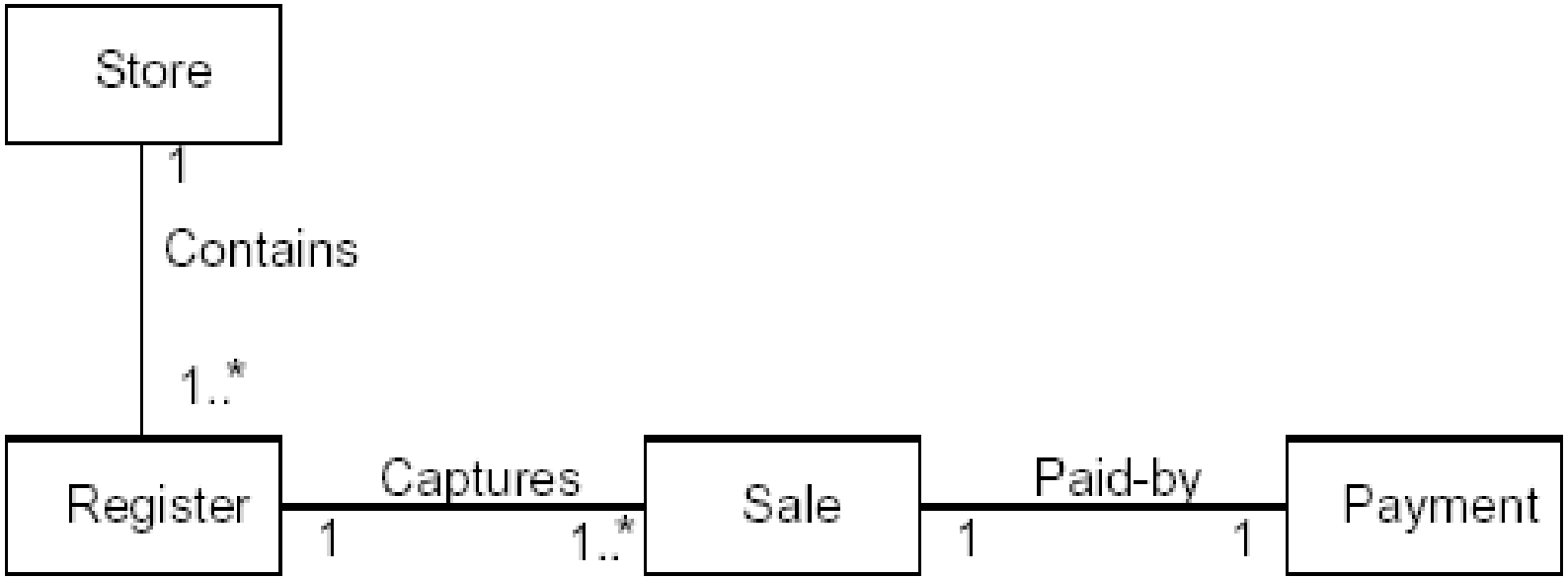
- Focus on those associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-know" associations).
- It is more important to identify *conceptual classes* than to identify associations.
- Too many associations tend to confuse a domain model rather than illuminate it. Their discovery can be time-consuming, with marginal benefit.
- Avoid showing redundant or derivable associations.

# Roles

- Each end of an association is called a **role.** Roles may optionally have:
  - name
  - multiplicity expression
  - navigability

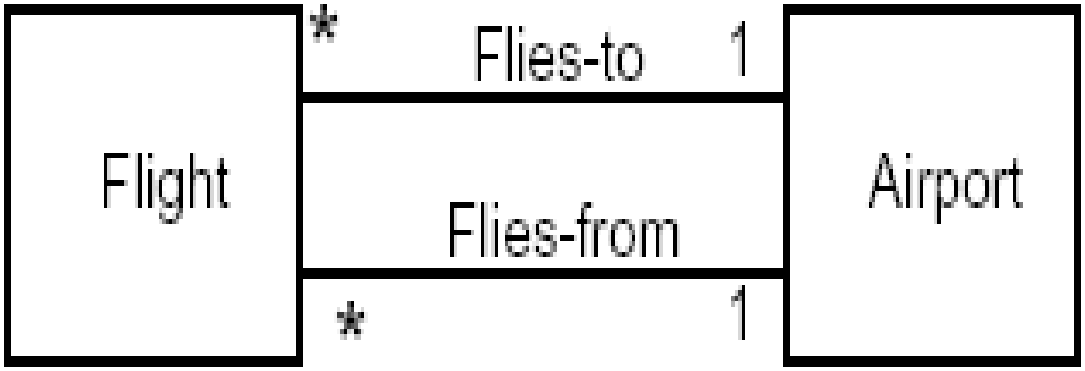
# Naming Associations

- Name an association based on a *TypeName-VerbPhrase-TypeName* format where the verb phrase creates a sequence that is readable and meaningful in the model context.



# Multiple Associations Between Two Types

- Two types may have multiple associations between them;
  - this is not uncommon.



# Associations and Implementation

- During domain modeling, an association is *not* a statement about data flows, instance variables, or object connections in a software solution;
- it is a statement that a relationship is meaningful in a purely conceptual sense—in the real world.

# Example

- The following sample of associations is justified in terms of a need-to-know.
  - It is based on the use cases currently under consideration

| | |
|---|---|
| *Register Records Sale* | To know the current sale, generate a total, print a receipt. |
| *Sale Paid-by Payment* | To know if the sale has been paid, relate the amount tendered to the sale total, and print a receipt. |
| *ProductCatalog Records ProductSpecification* | To retrieve an *ProductSpecification*, given an itemID. |

# Example-*Applying the Category of Associations Checklist*

| Category | System |
|---|---|
| A is a physical part of B | *Register — CashDrawer* |
| A is a logical part of B | *SalesLineItem — Sale* |
| A is physically contained in/on B | *Register — Store*<br>*Item — Store* |

| Category | System |
|---|---|
| A is logically contained in B | *ProductSpecification — Product-Catalog*<br>*ProductCatalog — Store* |
| A is a description for B | *ProductSpecification — Item* |
| A is a line item of a transaction or report B | *SalesLineItem — Sale* |
| A is logged/recorded/reported/captured in B | *(completed) Sales — Store*<br>*(current) Sale — Register* |
| A is a member of B | *Cashier — Store* |
| A is an organizational subunit of B | *not applicable* |
| A uses or manages B | *Cashier — Register*<br>*Manager — Register*<br>*Manager — Cashier, but probably not applicable.* |
| A communicates with B | *Customer — Cashier* |
| A is related to a transaction B | *Customer — Payment*<br>*Cashier — Payment* |
| A is a transaction related to another trans-action B | *Payment — Sale* |
| A is next to B | *SalesLineItem — SalesLineItem* |
| A is owned by B | *Register — Store* |

# Example

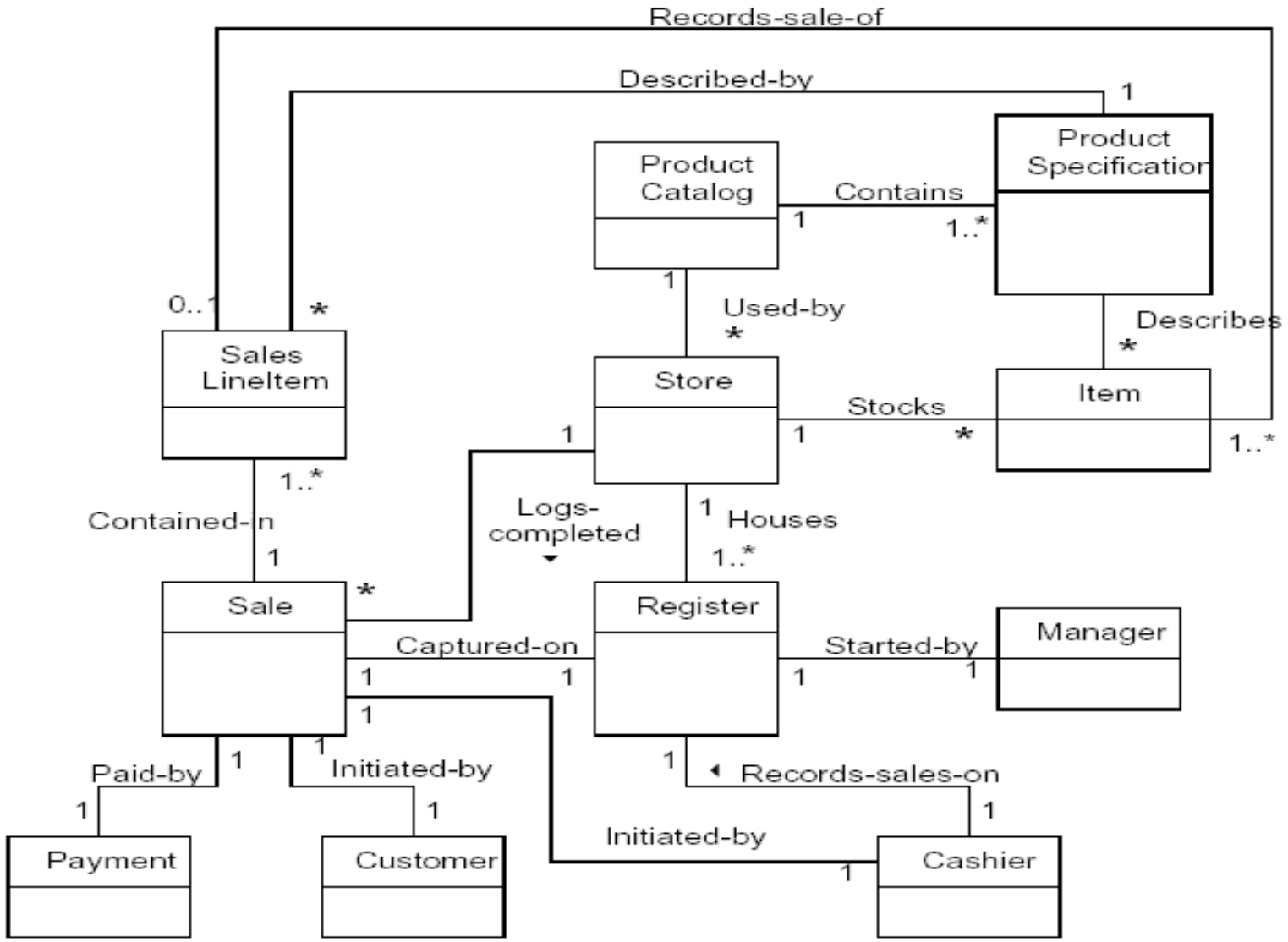



Figure 11.8 A partial domain model.

# Example

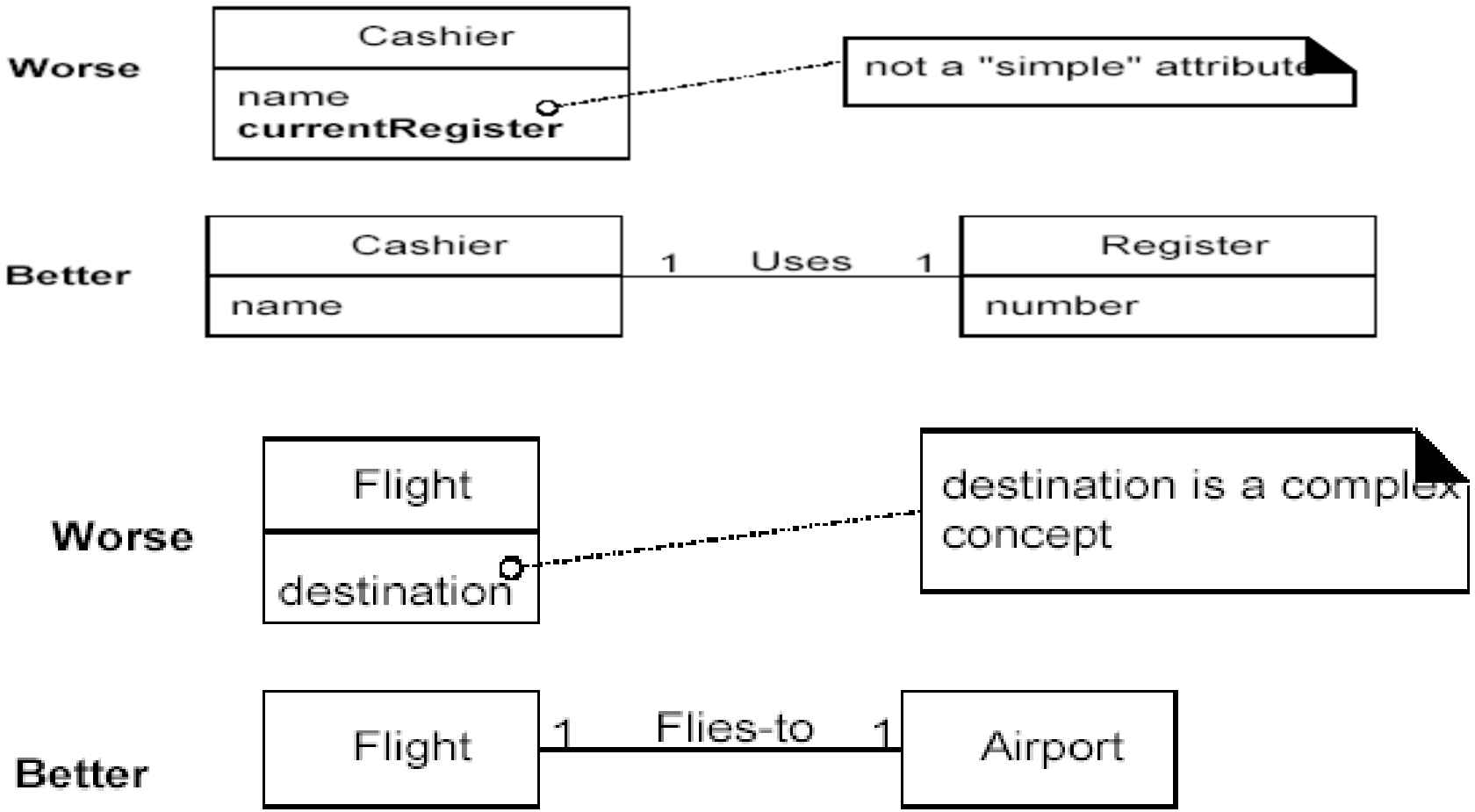
| Association | Discussion |
|---|---|
| *Sale Entered-by Cashier* | The requirements do not indicate a need-to-know or record the current cashier. Also, it is derivable if the *Register Used-by Cashier* association is present. |
| *Register Used-by Cashier* | The requirements do not indicate a need-to-know or record the current cashier. |
| *Register Started-by Manager* | The requirements do not indicate a need-to-know or record the manager who starts up a *Register*. |
| *Sale Initiated-by Customer* | The requirements do not indicate a need-to-know or record the current customer who initiates a sale. |
| *Store Stocks Item* | The requirements do not indicate a need-to-know or maintain inventory information. |
| *SalesLineItem Records-sale-of Item* | The requirements do not indicate a need-to-know or maintain inventory information. |

# Adding Attributes

- **Attributes vs. Associations**
- **Non-primitive Data Type Classes**
- **No Attributes as Foreign Keys**
- **Modeling Attribute Quantities and Units**

# Attributes VS. Associations

- **The attributes in a domain model should preferably be simple attributes or data types.**



# End

**Next we look at Object Oriented Design**