The Domain Model

Unlike other engineers, software engineers work on different areas, with various needs and business rules.

For example they develop software for airline companies, for banks, for embedded systems like car engines.

Therefore it is not sufficient to know about *software domain* , a software engineer also needs to know about the *problem domain* .

A domain model illustrates concepts in a problem domain (real-world).

UML class diagrams are used to present domain models.

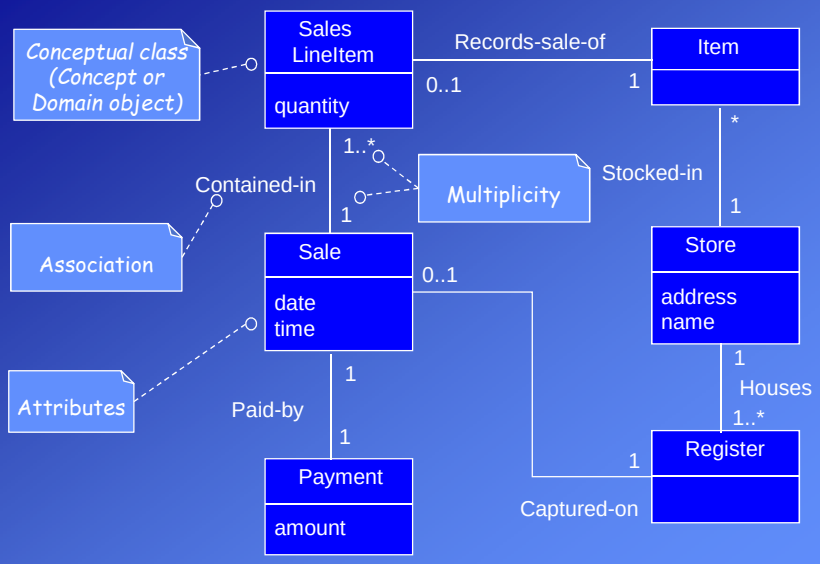
It may show three items:

- domain objects or conceptual classes
- associations between conceptual classes
- attributes of conceptual classes

Benefits of the domain model:

- It helps to understand the system.
- It acts as a source to define software classes in design level.

Example: A Partial Domain Model

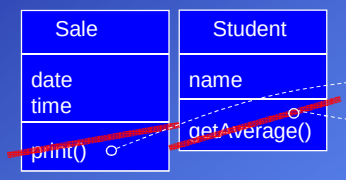


A domain model shows real-situation conceptual classes, not software classes.

Real-world concept:
OK for domain model

Software artifacts are not a part of domain model.

Avoid. Software artifact. May be included in design model

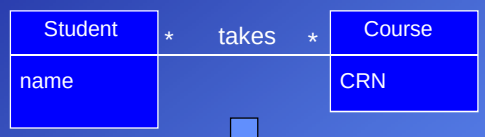


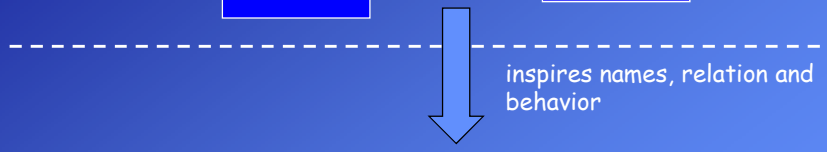
Avoid. Software classes.
Responsibilities will be assigned in design step.

Lower representational gap between real-world and software

Names and responsibilities of software classes in design level are inspired from conceptual classes in the domain model.

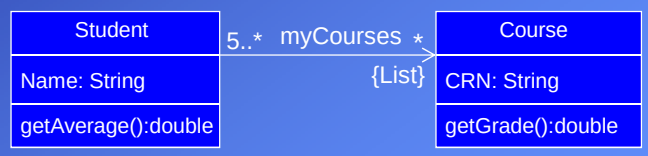
Domain model
Conceptual classes:





inspires names, relation and behavior

Design model
Software classes:



Domain (analysis) model and software (design) model will not be the same but similar.

How to Create the Domain Model?

1. Find the conceptual classes.
2. Add associations and attributes.
3. Draw them as classes in a UML class diagram.

How to Find Conceptual Classes?

Three strategies to find conceptual classes:

1. Reuse or modify existing models.

   If there is an existing model from a previous project it can be modified.

   There are also published domain models for many common domains, such as inventory, finance, health, and so forth.
2. Use a category list.

   You can define conceptual classes in your application domain by using the list that contains many common categories.
3. Identify noun phrases in the use cases.

Finding Conceptual Classes with Noun Phrase Identification

Identify the nouns and noun phrases in textual descriptions of a domain (use cases), and consider them as candidate conceptual classes or attributes.

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. Cashier enters **item identifier**.
4. System records **sale line item** and presents **item description**, **price**, and running **total**. 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:

7a. Paying by cash:

   1. Cashier enters the cash **amount tendered**.
   2. System presents the **balance due**.

.....

Eliminating unnecessary noun phrase

All noun phrases in use cases do not represent conceptual classes.

Following noun phrases should be eliminated:

1. Different noun phrases may represent the same conceptual class.
   For example, customer and user are redundant . Use "customer" because it is more descriptive.
2. Some noun phrases may refer to conceptual classes that are ignored in this iteration (for example, "accounting" and "commissions").
3. Some noun phrases may refer to attributes. Attributes should be basic data types such as, number, text.

This method can be used in combination with the "Conceptual Class Category List" technique.

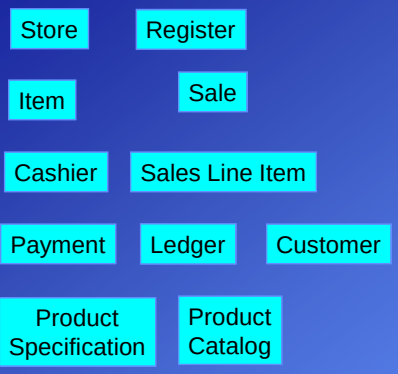
The Mapmaker Approach

A domain model is a kind of map of concepts or things in an application domain.

Make a domain model in the spirit of how a cartographer or mapmaker works:

- **Use the existing names** in the territory.
  Mapmakers do not change the names of cities on a map.
  Use the vocabulary of the domain when naming conceptual classes and attributes.
- **Exclude irrelevant features**.
  For example, in a physical map, borders of cities are not shown.
  Do not put classes or attributes on the model if they do not have any obvious noteworthy role. For example; keyboard, the age of cashier.
- **Do not add things that are not there**.
  A mapmaker does not show things that are not there, such as a mountain that does not exist.
  Similarly, the domain model should exclude things *not* in the problem domain under consideration. For example; the owner of the store.

Example: Conceptual classes of NextGen POS system



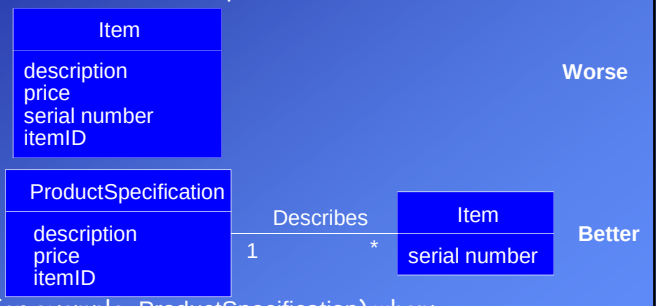
- Register
- Item
- Store
- Sale
- Sales Line Item
- Cashier
- Customer
- Ledger
- Payment
- Product Catalog
- Product Specification

There is no such thing as a "correct" list.
However, by following the identification strategies, different modelers will produce similar lists.

Need for description (specification) classes

Assume that information about physical items in a store are written on these items; such as serial number and price.
It seams logical because these data are attributes of these items.
But when all items are sold out some data may be lost.

In such systems it is necessary to keep these data in separate description classes.

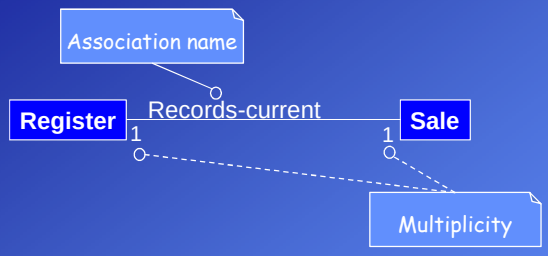


Add a description class (for example, ProductSpecification) when:

- We need the description about an item, even the examples of this item currently does not exist in the system.
- Deleting instances of things they describe (for example, Item) results in a loss of information that needs to be maintained.
- It reduces redundant or duplicated information.

Associations

An **association** is a relationship between classes that indicates some meaningful connection .



- Because the domain model displays conceptual classes in real-world, associations also refer to real situations.
- Associations are not function calls between software classes.
- Associations in domain model are mostly bidirectional.

Example: Register records current Sale. (From left to right)

Current Sale is recorded by Register. (From right to left)

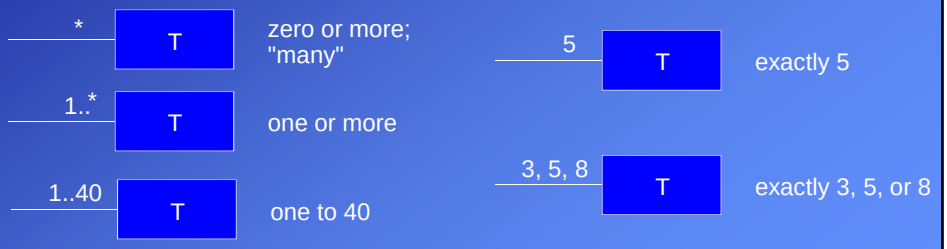
Multiplicity

Multiplicity defines how many instances (objects) of a class A can be associated at the same time (at a particular moment) with one instance of a class B.

Left to right: One student takes at least one or more courses.

Right to left: One course is taken at least by five or more students.

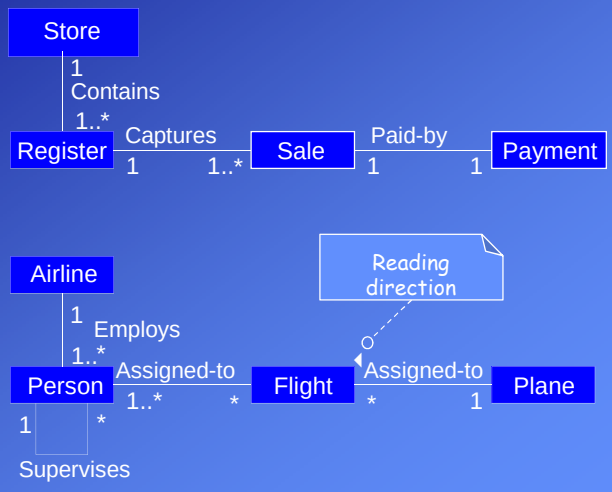
Multiplicity numbers are obtained from the requirements of the user.



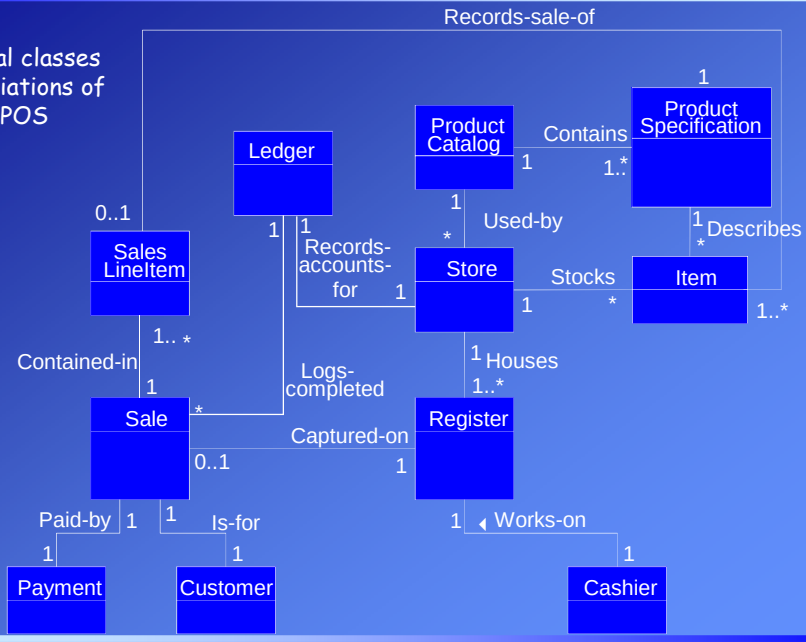
Finding Associations

Associations can be defined
1. by using the common associations list
2. by using verbs in use cases

Examples:



Example:
Conceptual classes and associations of NextGen POS system



Attributes

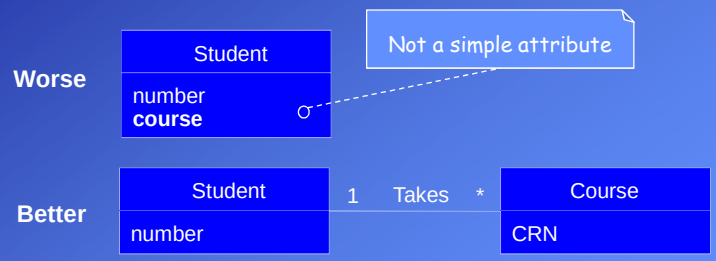
Identify attributes of conceptual classes (real-world attributes not software data)

Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information.

For example; register number of a student, date and time of a sale.

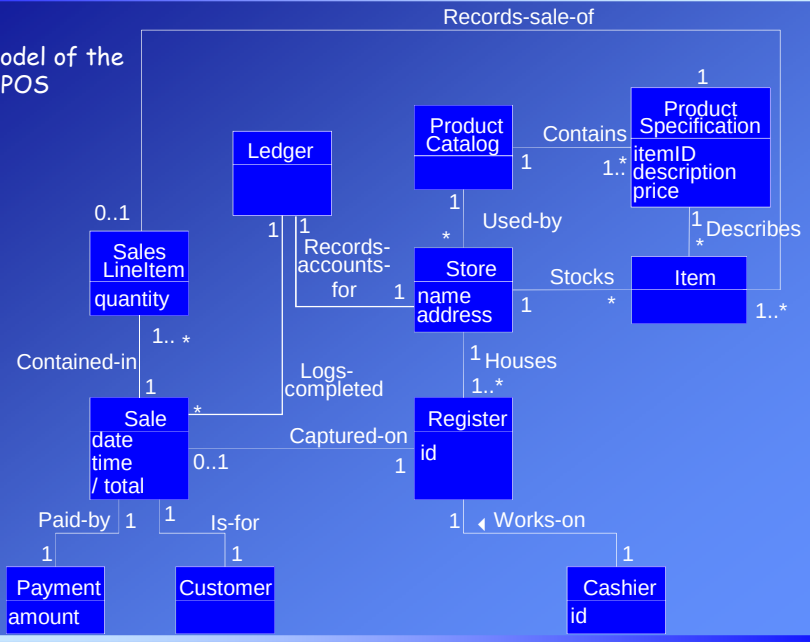
Attribute types should be "primitive" data types, such as numbers , characters, and booleans.

The type of an attribute should not normally be a complex domain concept.



If a concept has its own properties and behavior, then its is not a simple attribute it is a separate conceptual class.

Example:
Domain model of the NextGen POS system



Operation Contracts

Use cases and domain model are usually sufficient to understand requirements of the user and the expected features of the system under discussion.

Now it is possible to start with design.

However, sometimes a more detailed or precise description of system behavior is necessary.

In that case operation contracts can be written for necessary system operations in use cases.

The most important part of a contract is "postconditions".

The **postconditions** describe changes in the state of objects in the domain model when the related operation has finished.

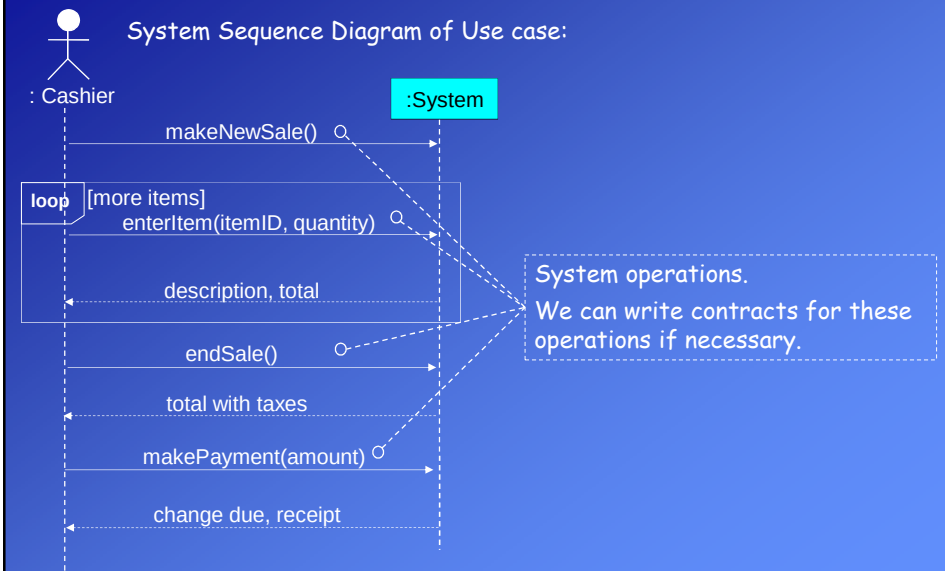
Postconditions are divided into three categories:

- Instance creation and deletion.
- Attribute change of value.
- Associations (UML links) formed and broken.

Remember, we are still in the real-world (application domain). We are not talking about software objects or attributes.

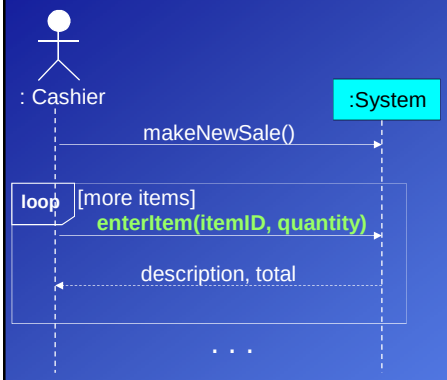
Examples: The given examples are related to the Use Case UC1: Process Sale of NextGen POS.

System Sequence Diagram of Use case:



Example: enterItem

The statement in the use case "Cashier enters item identifier." can be a complicated operation. Therefore we write a contract for this operation.



All objects (for example sli), mentioned in contracts are related to the domain model (3.16).
We are not writing the program.

Contract CO2: enterItem

Operation:
enterItem(itemID: ItemID, quantity: integer)

Reference: Use Cases: Process Sale

PreCond.: There is a sale underway

PostConditions:
- A SalesLineItem instance sli was created (instance creation)
- sli was associated with the current Sale (association formed)
- sli.quantity became quantity (attribute modification)
- sli was associated with a Product Spec. based on itemID match (association formed)

How two write postconditions?

Postconditions are not actions to be performed during the operation;

rather, they are observations about the domain model objects that are true when the operation has finished.

What happened to the objects in the system (real-world) after the operation?

We are still interested in **what** happened, not **how** it is performed.

How this contracts are realized is the issue of the design level.

Express postconditions in the **past tense** to emphasize they are observations about state changes.

Analogy: The Stage of a theater (Taken from Larman)

The system and its objects are presented on a theatre stage.

1. Before the operation, take a picture of the stage.
2. Close the curtains on the stage, and apply the system operation (background noise of clanging, screams, and screeches…).
3. Open the curtains and take a second picture.
4. Compare the before and after pictures, and express as postconditions the changes in the state of the stage (A SalesLineItem was created…).

Now we know the changes but not how (by whom) they are made.

Example: endSale

Assume that in NextGen POS system completed sales are not deleted, they are only marked as "completed" and logged in the system.

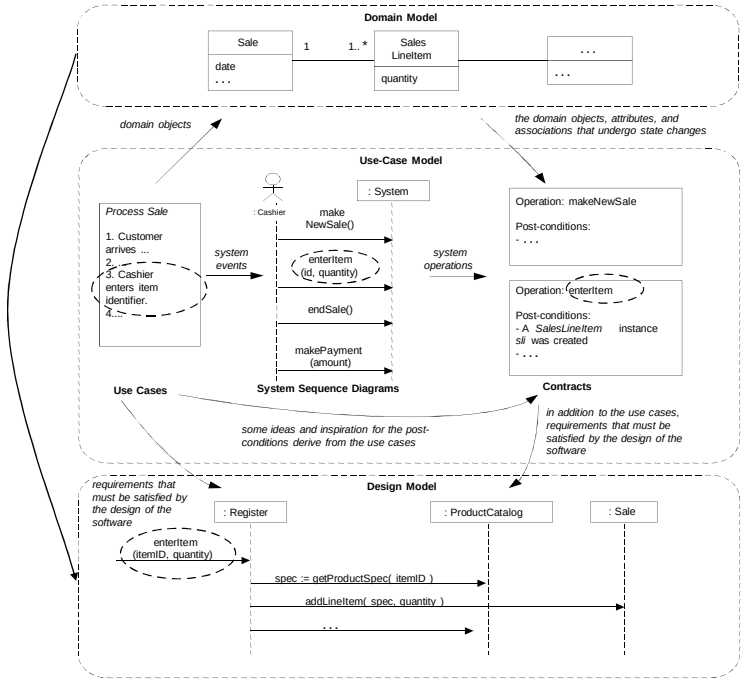
Contract CO3: endSale

Operation:	endSale()
Cross References:	Use Cases: Process Sale
PreConditions:	There is a sale underway
PostConditions:	- Sale.isComplete became true (attribute modification)

Sale
isComplete: Boolean date time

This attribute was not in the domain model

We discovered it during writing the contract.



3.22