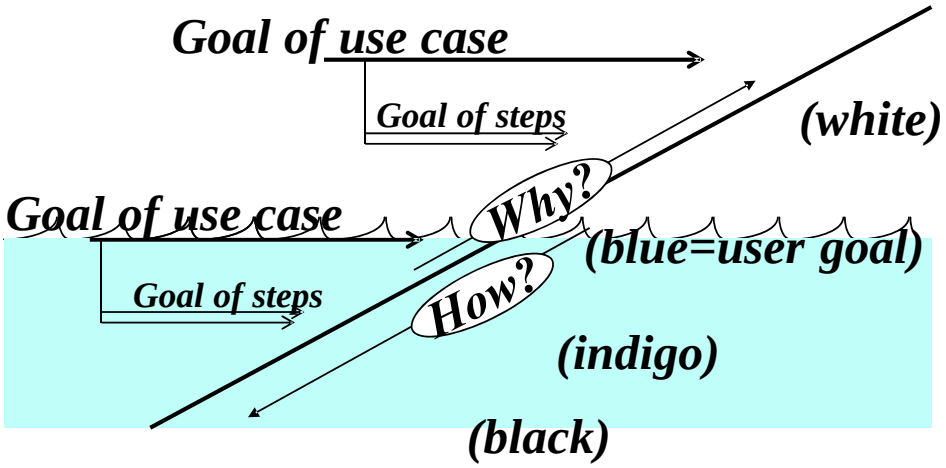
Object-Oriented Analysis and Design using UML and Patterns

Use Cases & Scenarios

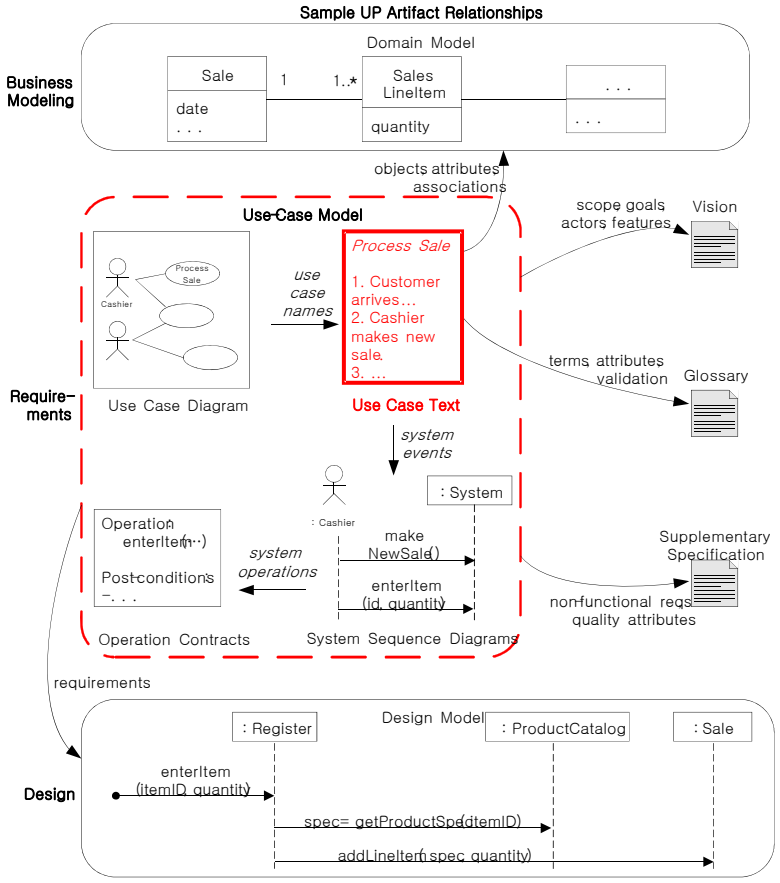


Use Cases & Scenarios

Objectives

- Identify and write use cases.
- Understand use case types and formats.
- Apply tests to identify suitable use cases.
- Relate use case analysis to iterative development.

Where are we?



Requirements

- Requirements are *capabilities* and *conditions* to which the system – and more broadly the project – must conform.
- Sloppy requirements specification and management is a major risk in software development.
  - Therefore, worth the effort to do very well.
- In the UP, one of the promoting best practices is *manage requirements*.
  - *A systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system in the context of inevitably changing and unclear stakeholder's wishes.*

Useful Classification

- Functional (i.e., behavioral) requirements *vs.* Non-functional requirements
- Most of the non-functional requirements are collectively called as *quality attributes* or *quality requirements*
  - the "-ilities" of a system

Artifacts	Requirements
Use-Case Model	Functional requirements, and other related requirements are explored and recorded
Supplementary Specifications	Non-functional requirements are recorded.
Vision	Summarizes high-level requirements
Glossary	Encompasses the concept of *data dictionary* which records requirements related to data, such as validation rules, acceptable values, and so forth.

Motivation for Use Cases

- We need a tool to analyze, record, and improve functional requirements
  - What do we really mean with each requirement
- Developers must know exactly what they are going to implement
  - Need a tool to collect and analyze functional requirements
  - Need a tool to model the functionality of the system as early as possible
- A client must understand what will be implemented
  - He can study the system before implementation
  - Decisions can be agreed
  - Final product can be checked against what was agreed
- A solution - *use case*

Goals and Stories

- Stakeholders, e.g., clients and end users, have *goals* (aka, *needs*) and want the system to help meet them.
  - Stakeholder: someone or something with a vested interest in the behavior of the SuD
- The most effective way to capture the *goals* and system's *functional requirements* is writing *use cases*.
- Informally, use cases are *stories of using a system to meet goals*.
- Because use cases are simple and familiar, it makes for clients and end users to contribute to their definition or evaluation.

Use Case Example: Process Sale

Main Success Scenario (or **Basic Flow**)

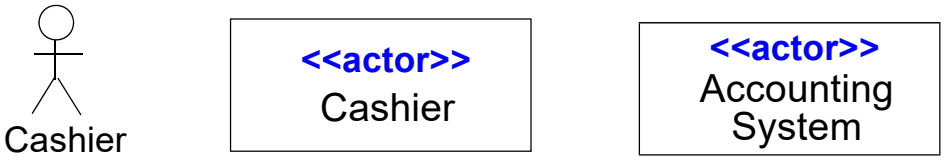
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

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).

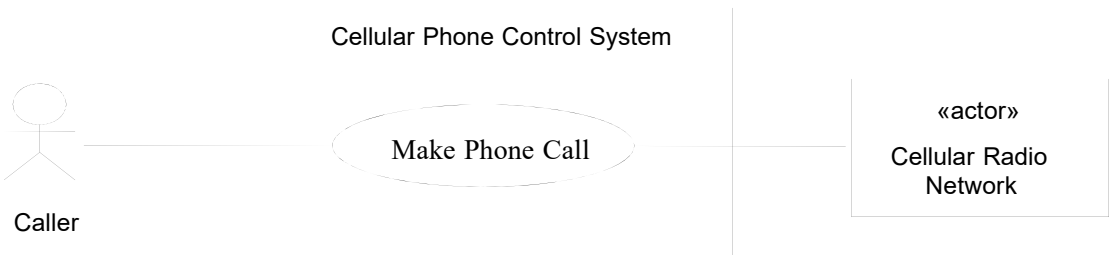
Actors

- Basic Definition:
  - An *actor* is anything outside SuD that interacts with SuD.
  - E.g., a person, a hardware device, organization or another system.
- In case of a person actor, it represents a *role* that the person plays when interacting with these use cases.



Types of Actors

- Primary actors
  - Has user goals fulfilled through using services of the SuD
  - Why identify? To find user goals, which drive the use cases
- Supporting actors
  - Provides a service to the SuD
  - Why identify? To clarify external interfaces and protocols



Scenarios

- A *scenario* (aka, *use case instance*) is a ***specific*** sequence of actions and interactions between actors and the SuD to meet a ***common*** goal.
- It is one particular story of using a system, or one path through the use case.
  - E.g., the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit card transaction denial.

Use Cases

- Informally, a use case is a collection of related success and failure scenarios that describes actors using a system to support a common goal.
- UP Definition: A use case is a set of use-case instances, where each instance is a sequence of actions that yields an *observable result of value* (aka, benefits) to a particular actor.
- Use cases document the behavior (i.e., functional requirements) of the system *from the user's point of view.*
- A software product implements a set of use cases.
- First introduced by Ivar Jacobson in the early 90's.

Examining the Goals the system supports makes good functional requirements.

“Place an order.”

“Get money from my bank account.”

“Get a quote.”

“Find the most attractive alternative.”

“Set up an advertising program.”

- Goals summarizes system function in understandable, verifiable terms of use.

A use case contains the set of possible scenarios for achieving a goal.

- All the interactions relate to the same goal of the same primary actor.
- The use case starts at the triggering event and continues until the goal is delivered or abandoned, *and* the system completes its responsibilities w.r.t. the interaction.

The use case pulls goal & scenarios together, Each scenario says how 1 condition unfolds.

- The use case name is the goal statement:

  "Order product from catalog"

Scenario (1): Everything works out well …

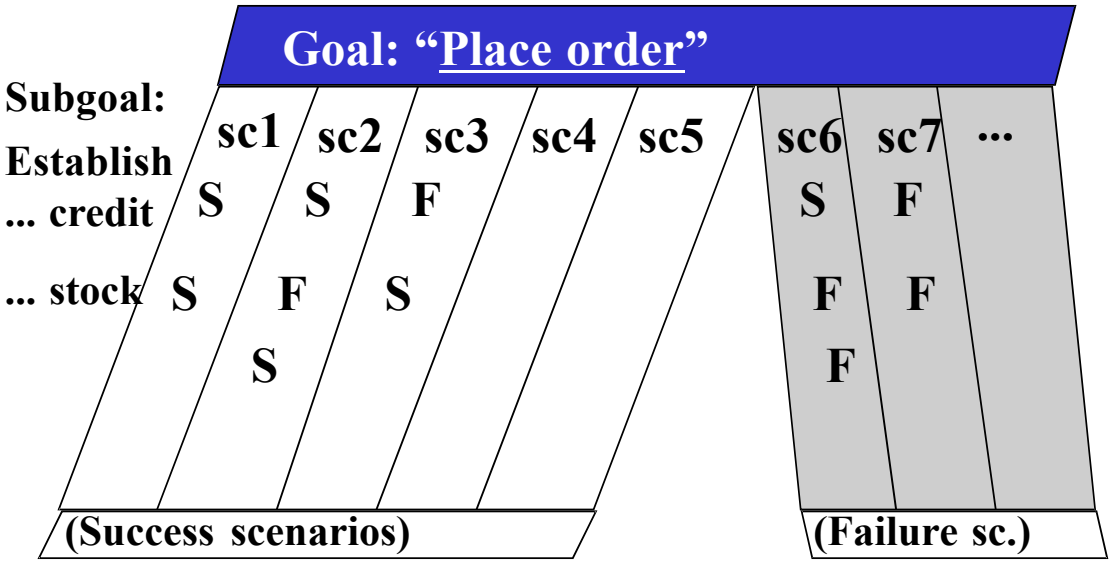
Scenario (2): Insufficient credit …

Scenario (3): Product out of stock …

- Use case is goal statement plus the scenarios.

(Note the grammar: active verb first)

The collected scenarios are like stripes on trousers, with success and failure legs.



Users, executives and developers appreciate seeing requirements in the form of goals.

- Users:
  - *"We can understand what these mean"*
  - *"You mean we are going to have to … ?"*
- Executives:
  - *"You left one thing here …"*
- Developers:
  - *"These are not just a pile of paragraphs!"*

Business UC vs. System UC

- Business Usecase
  - Describe the operations of the business.
- System Usecase
  - Describes the functional requirements for the SuD.

Black-Box vs. White-Box UCs

- Black-Box Use Case
  - Do not describe the internal workings of the system, its components, or design
  - Describes *what* the system must do (the functional requirements) w/o deciding *how* it will do it (the design)
- White-Box Use Case
  - Show the internal workings of the system, its components, or design
  - Business process designers may write white-box business usecase to show how the company or organization runs its internal processes.
  - The technical development team may write the same to document the operational context for the SuD they are about to design, and might write white-box system use cases to document the workings of the system they just designed.

Use Case Formats

- Brief format
  - Terse one-paragraph summary, usually of the main success scenario.
  - Created in early requirements phase to understand the degree of complexity and functionality in a system.
- Casual format
  - Informal paragraph format. Multiple paragraphs that cover various scenarios.
- Fully dressed format (One-column or Two-column)
  - The most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

Brief Format UC Example

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Casual Format UC Example

Handle Returns

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item …

Alternate Scenarios:

If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).

If the system detects failure to communicate with the external accounting system, …

Two-Column Format UC

Use Case UC1: Process Sale

Primary Actor: ...
... as before ...

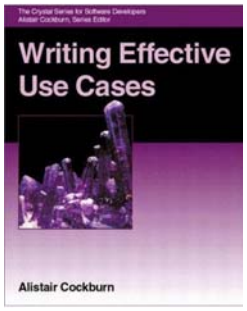
Main Success Scenario:

Actor Action (or Intention)	System Responsibility
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. Records each sale line item and presents item description and running total.
Cashier repeats steps 3-4 until indicates done.	
	5. Presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.	
7. Customer pays.	
	8. Handles payment.
	9. Logs the completed sale and sends information to the external accounting (for all accounting and commissions) and inventory systems (to update inventory). System presents receipt.
...	...

The use case is a behavioral part of the contract between various stakeholders.

Every sentence in a use case describe

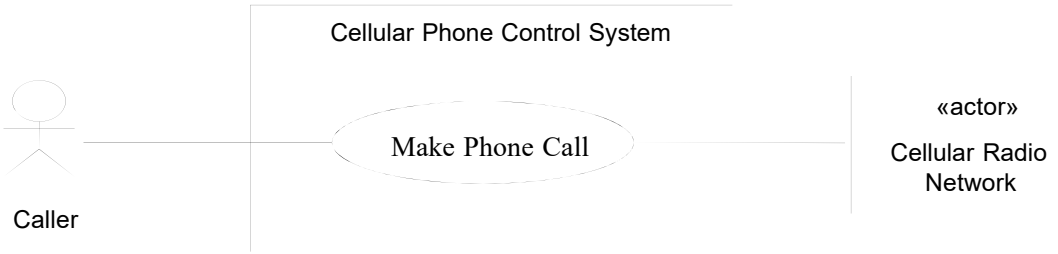
- an action or an *interaction* between two *actors with goals*, or
- what the system must do internally to protect the *stakeholders with interests*.



Definition of Actors in Actors with Goals Model

- Basic Definition
  - An *actor* is anything outside SuD that interacts with SuD.
  - E.g., a person, a hardware device, organization or another system.

- **Extended Definition**
  - **An *actor* is anything with behavior, including the SuD itself when it calls on the services of other actors.**
    - Only when we view use cases from *Actors with Goals Model* perspective

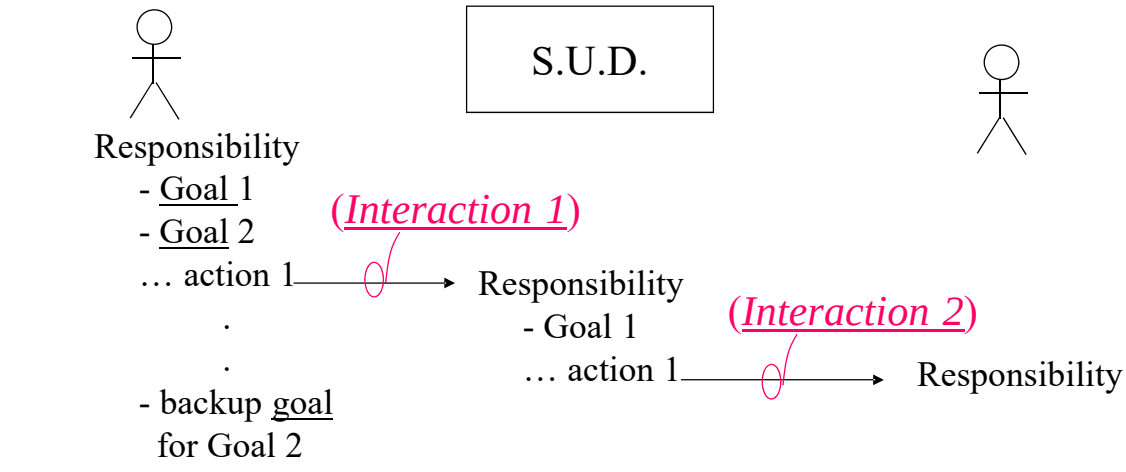


The basic model of use cases is that Actors interact to achieve their goals.

Primary Actor
person or system with goal for s.u.d.

System under Design
could be any system

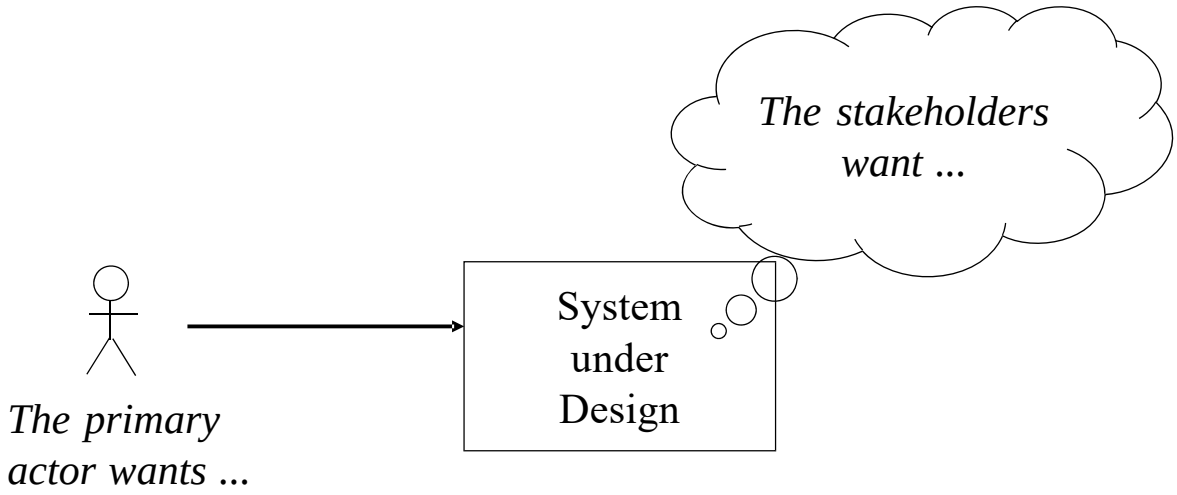
Supporting Actor
other system against which s.u.d. has a goal



Types of Actors for Stakeholders with Interests Model

- Primary actors
  - Has user goals fulfilled through using services of the SuD
  - Why identify? To find user goals, which drive the use cases
- Supporting actors
  - Provides a service to the SuD
  - Why identify? To clarify external interfaces and protocols
- **Offstage actors**
  - **Has an interest in the behavior of the use case, but is not primary or supporting actors**
  - **Why identify? To ensure that all the necessary interests are captured and satisfied.**

The system protects the interests of all the stakeholders.



A use case can be viewed as a behavioral contract between stakeholders with interests

- The *Actors with Goals* model explains how to write sentences in the use case, but it *does not cover the need to describe the internal behavior* of the system under discussion.
- Therefore, we need to extend the Actors and Goals model to *Stakeholders with Interests* model.
- The Stakeholders and Interests model identifies what to include in the use case.

To satisfy the interests of the stakeholders, we need to describe three sorts of actions.

- An action or interaction between two actors (to further a goal)
- A validation (to protect the interest of one of the stakeholders)
- An internal state change (on behalf of a stakeholder also to protect or further an interest of a stakeholder)

The scenario ends with all the interests of the stakeholders are satisfied or protected.

UC: Fully Dressed Format

UC1: Process Sale

- Primary Actor
- Stakeholders and interests
- Preconditions
- Success Guarantee (Postconditions)
- Main Success Scenario (or Basic Flow)
- Extensions (or Alternative Flows)
- Special Requirements
- Technology and Data Variations List
- Frequency of Occurrence
- Open Issues

Primary Actor

- The stakeholder who or which initiates an interaction with the SuD or calls upon system services to achieve a goal.

Primary Actor: Cashier

Stakeholders and Interests

- The use case, as the contract for behavior, captures all and only the behaviors related to satisfying the stakeholders' interests.
- Therefore, this answers the question: What should be in the use case? The answer is: That which satisfies all the stakeholder's interests.
- This section is important because it provides a thorough and methodical procedure for discovering and recording all the required behaviors.

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 updates
- …

Preconditions and Postconditions

- *Preconditions* state what *must always* be true before beginning a scenario in the use case. They are conditions that are assumed to be true.
  - Typically, a precondition implies a scenario of another use case that has successfully completed.
- *Postconditions* state that what must be true on successful completion of the use case, regardless of its path.
  - The postconditions should meet the needs of all stakeholders.

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

Main Success Scenario

- *Happy Day Scenario* or *Basic Flow*
- It describes typical success path that satisfies the interests of the stakeholders.
- It records three kinds of actions:
  1. An interaction between actors.
  2. A validation
  3. A state change by the system
- Step one of use case normally indicates the trigger event that starts the use case.
- *Defer all conditional and branching to the Extensions section.*

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. …

Cahier repeats steps 3-4 until indicates done.

5. …

Extensions (or Alternative Flows)

- Extensions indicate all the other scenarios or branches, both success and failure.
- An extension has two parts: ***condition*** and ***the handling***
- **Guideline**: Write the condition as something that can be *detected* by the system or an actor. To contrast:
  - 5a. System detects failure to communicate with external tax cal. system service:
  - 5a. External tax cal. System not working:

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.
3-6a. Customer asks Cahier to remove an item from the purchase:
    1. Cashier enters the item identifier for removal from the sale.
    2. System displays updated running total.

Extensions (Cont'd)

- At the end of extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (such as by halting the system.)
- If a particular extension is quite complex, a separate use case may be written.

7b. Paying by credit:
1. Customer enters their credit card information.
2. System requests payment validation from external Payment Authorization Serive system.
2a. System detects failure to collaborate with external system.
1. System signals error to Cashier.
2. Cashier asks Customer for alternate payment.
3. …

*a. At any time, System crashes:
In order to support recovery and correct accounting, ensure all transaction sensative state and events can be recovered at any step in the scenario.
1. Cahier restarts the System, log in, and requests recovery of prior state.
2. System reconstructs prior state.

A scenario refers to low-level goals: subordinate use cases or common functions.

UC 4: Place an order
1. Identify customer (UC 41)
2. ...

Note the active verbs!

UC 41: Identify customer
1. Operator enters name.
2. System finds near matches.

- Extensions:
2a. No match found: ...

Special Requirements

- Non-functional requirements, quality attributes, and constraints which relates specifically to a use case are recorded with the use case.

Special Requirements:
- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds within 90% of time.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 2 and 6.

Technology and Data Variations List

- This section records technical variations in how something must be done, but not what.
  - E.g., I/O technologies
- This section also records variations in data scheme.
  - E.g., UPCs or EANs in bar code symbology
- Suggestion: This section should *not* contain multiple steps to express varying behavior for different cases. If that is necessary, say it in the Extensions section.

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.

Key four concepts that apply to every sentence in UC and to the UC as a whole

- Level: Why do we want this goal?
  - Enter $ amount, to ***get $***, to buy lunch
  - ("subfunction" vs. "user" vs. "summary" goal)
- Scope: Which system boundary do we mean?
  - The panel is part of the ***ATM***, is part of the bank.
  - ("internal" vs. "system" vs. "organization/corporation")
- Detail: Do we describe intent, or action detail?
  - Hit number buttons to ***enter $ amount.***
  - ("dialog description" vs. "semantic / intent")
- Primary Actor: Who has the goal?

Which of these is a valid use case?

- Negotiate a Supplier Contract
- Handle returns
- Log In

All of these can be valid use cases at different levels, Depending on the system boundary, actors, and goals.

What is a useful level to express use cases for application requirements analysis?

- Guideline: The *EBP* (or *User Goals*) Use Case
  - For requirements analysis for a computer application, focus on use cases at the level of **elementary business processes** (EBPs).
- EBP is defined as
  - *A task performed by one person in one place at one time*, in response to a business event, which adds measurable business value and leaves the data in a consistent state.

A Common Mistake with Use Cases

- A use case is a relatively large end-to-end description that typically includes many steps or transactions
  - It is not normally an individual step or activity in a process.

Every sentence at every level is a goal. Use cases are one sentence style repeated.

- Goal: “Customer places an order.”

how *why*

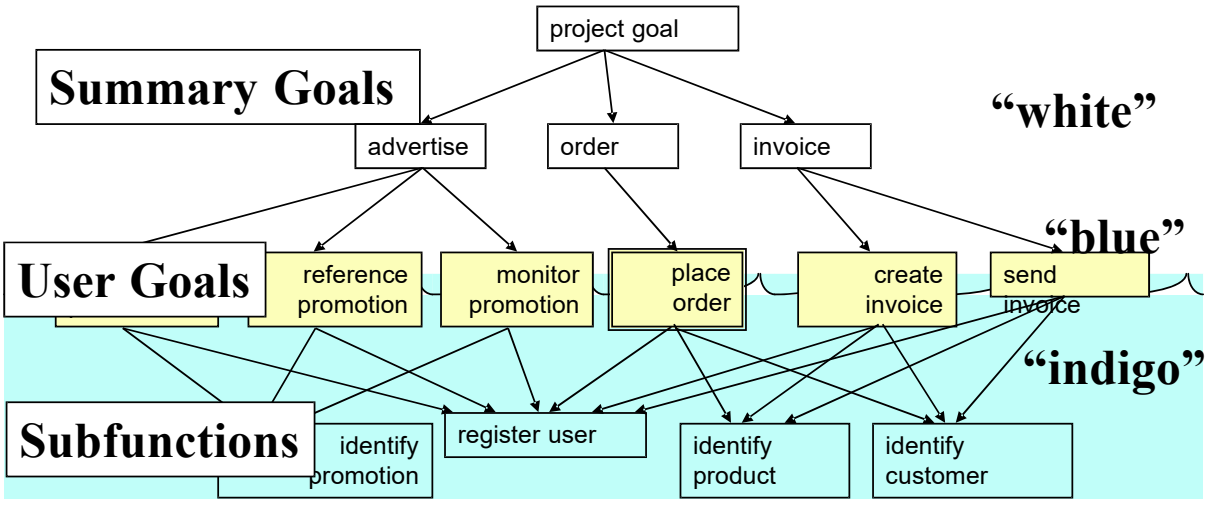
- Step: “Customer prepays for the order.”

how *why*

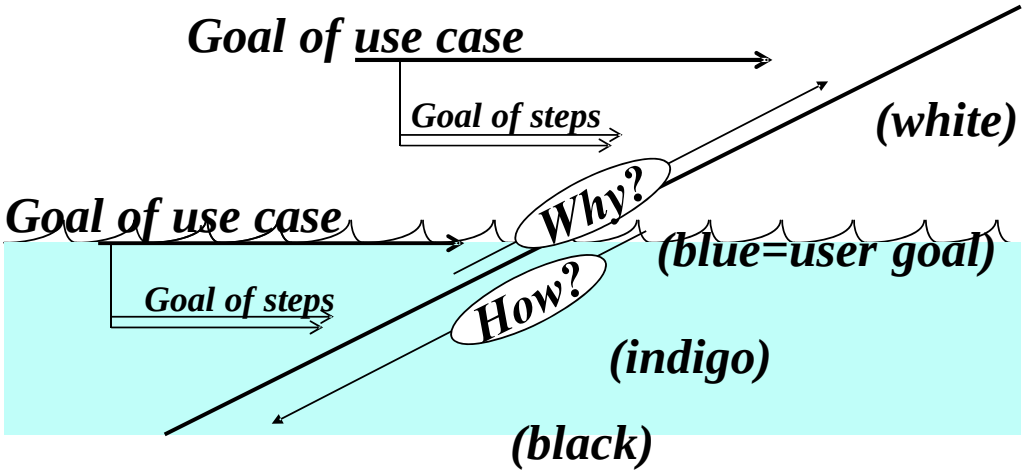
- Substep: “Customer gives credit card number.”

Summary-, User-, and Subfunction Goals link together as a graph.

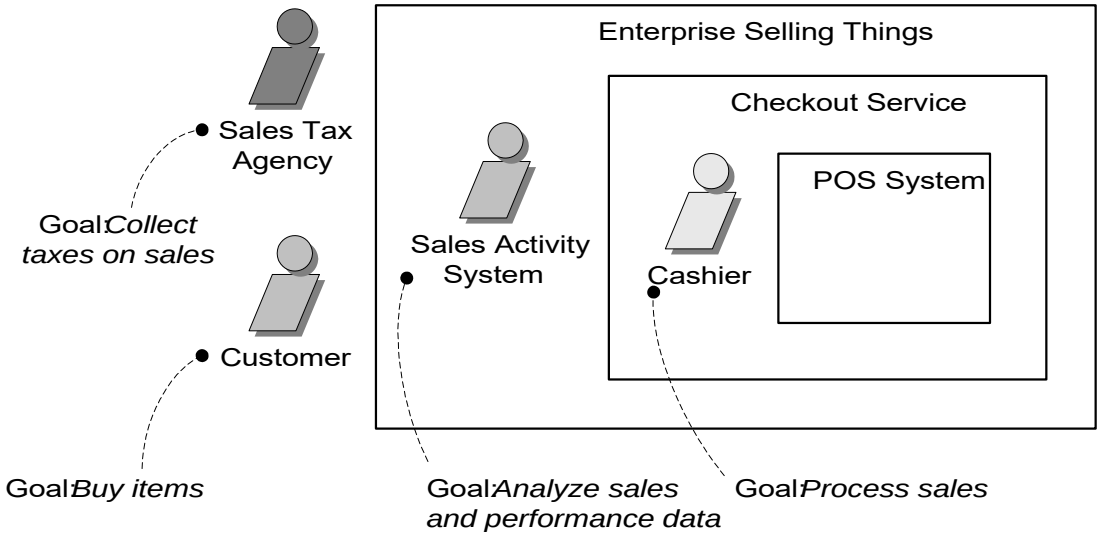
- Sailboat image: User goals are at sea level.



The use case goal level is higher than the steps. They white to blue, indigo, black.



Primary Actors and goals at different boundaries



Basic Procedure

1. Choose the system boundary.
2. Identify the primary actors.
3. For each primary actor, identify their user goals.
4. Define use cases that satisfy user goals; name them according to their goal.

Step 1: Choosing the System Boundary

- Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
- Choosing the right system boundary helps to find what is outside, i.e., primary actors and supporting actors.
- For our case study, the POS system itself is the system under design.

Step 2 and 3: Finding Primary Actors and Goals

- What computers, subsystems and people will drive our system?
  - An actor is anything with behavior.
- What does each actor need our system to do?
  - Each need shows up as a trigger to our system.
- Result: a list of use cases, a sketch of the system.
  - Short, fairly complete list of usable system function.

Reminder Questions to Find Actors

- Primary actors and supporting actors are always external to SuD. To find actors, look for things in the categories of people, other software, hardware devices, data stores, or networks.
- The following questions might also be useful to find actors:
  - Who uses the system?
  - Who installs the system?
  - Who starts up and shuts down the system?
  - Who maintains the system?
  - What other systems use this system?
  - Who gets information from this system?
  - Who provides information to the system?
  - Does anything happen automatically at a preset time?

Reminder Questions to Find Goals

- In the UP, a use case is always started by a primary actor. But, it is sometimes useful to initiate use cases from inside the system (e.g., timer).
- Actor-based
  - Identify the actors related to a system or organization.
  - For each actor, identify the process they initiate or participate in.
- Event-based
  - Identify the external events that a system must respond to.
  - Relate the events to actors and use cases.
- To identify goals and use cases, ask yourself:
  - What functions will the actor want from the system?
  - Does the system store information? What actors will create, read, update, or delete that information?
  - Does the system need to notify an actor about changes in its internal state?
  - Are there any external events that the system must know about? What actor informs the system about those events?

The Actor-Goal List
(Part of the Vision Artifact)

Actor	Goal	Priority
Cashier	Process Sales Process Rentals Handle Returns Cash In Cash Out …	1 2 1 3 3 …
Manager	Start Up Shut Down …	3 3 …
System Administrator	Manage Users Manage Security …	3 2 …
Sales Activity System	Analyze Sales Activity …	2 …
…	…	…

Ranking and Iteration Planning

Use cases need to be ranked, and high ranking use cases need to be tackled early.

Organize requirements and iterations by risk, coverage, and criticality.

- Risk
  - Both technical complexity and other factors, such as uncertainty of effort or usability.
- Coverage (or Architectural Significance)
  - Implies that all major parts of the system are at least touched on in early iterations – perhaps a "wide and shallow" implementation across many components.
  - Integrating exiting components
- Criticality
  - Refers to functions of high business value

Step 4: Define Use Cases

- In general, define one EBP-level (or user goals) use case for each user goal.
- Collapse CRUD separate goals into one CRUD use case, idiomatically called *Manage <X>*.
- Name use cases starting with an active verb.

How to do it: For each use case … (1). Write the simple case: goal delivers.

- The *main success scenario*, the happy day case.
  - Easiest to read and understand.
  - Everything else is a complication on this.
- Capture each actor's intent and responsibility, from trigger to goal delivery.
  - Say what information passes between them.
  - Number each line.
- Result: readable description of system's function.

How to do it: (2). Write failure conditions as extensions.

- Usually, each step can fail.
- Note the failure condition separately, after the main success scenario.
- Result: list of alternate scenarios.

How to do it: For each failure condition, (3). Follow the failure till it ends or rejoins.

- Recoverable extensions rejoin main course.
- Non-recoverable extensions fail directly.
- Each scenario goes from trigger to completion.
  - "Extension" are merely a writing shorthand.
  - Can write "if" statements.
  - Can write each scenario from beginning to end.
- Result: Complete use case.

Essential vs. Concrete Use Cases

- Essential use cases are expanded use cases that are expressed in an ideal form that remains relatively free of technology and implementation details.

1. The Customer identifies them-Selves.
2. Presents options.
3. And so on.
4. And so on

- Concrete use cases concretely describes the process in terms of its real current design, committed to specific input and output technologies, and so on.

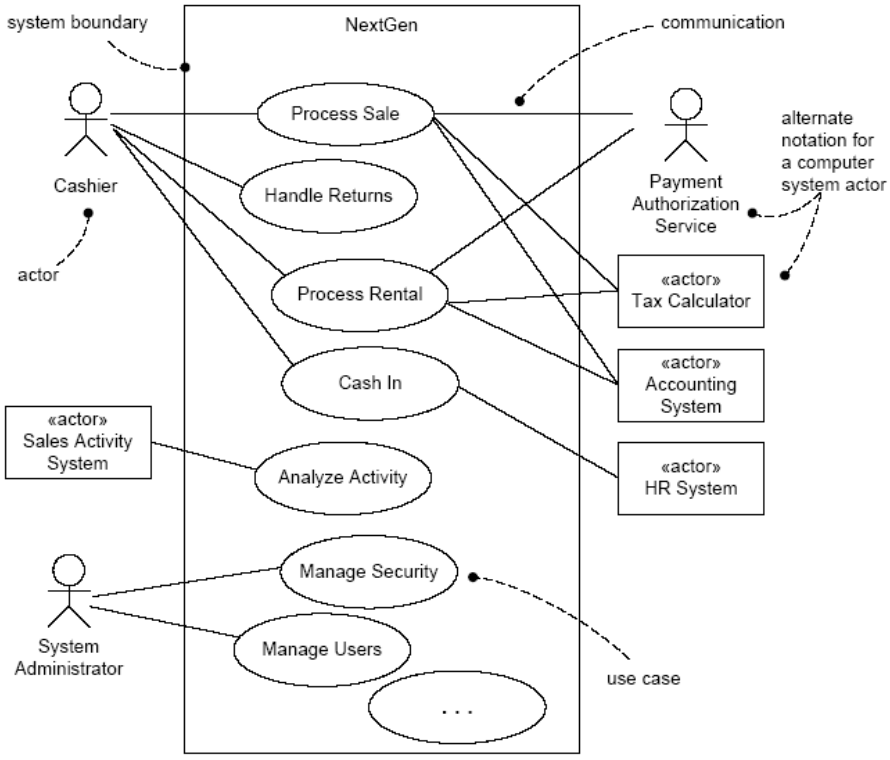
1. The Customer inserts their card.
2. Prompts for PIN.
3. Enters PIN on keypad.
4. Display options menu.
5. And so on.
6. And so on.

Use Case Diagram

- ***A use case diagram is a diagram that shows a set of use cases and actors and their relationships.***
  - Which actors carry out which use cases
  - Which use cases include/extends other use cases
- UML modeling elements
  - Use Cases
  - Actors
  - Dependency, generalization, and association relationships

Use case diagrams and use case relationships are secondary in use case work.
Use cases are text documents.
Doing use case work means to write text.

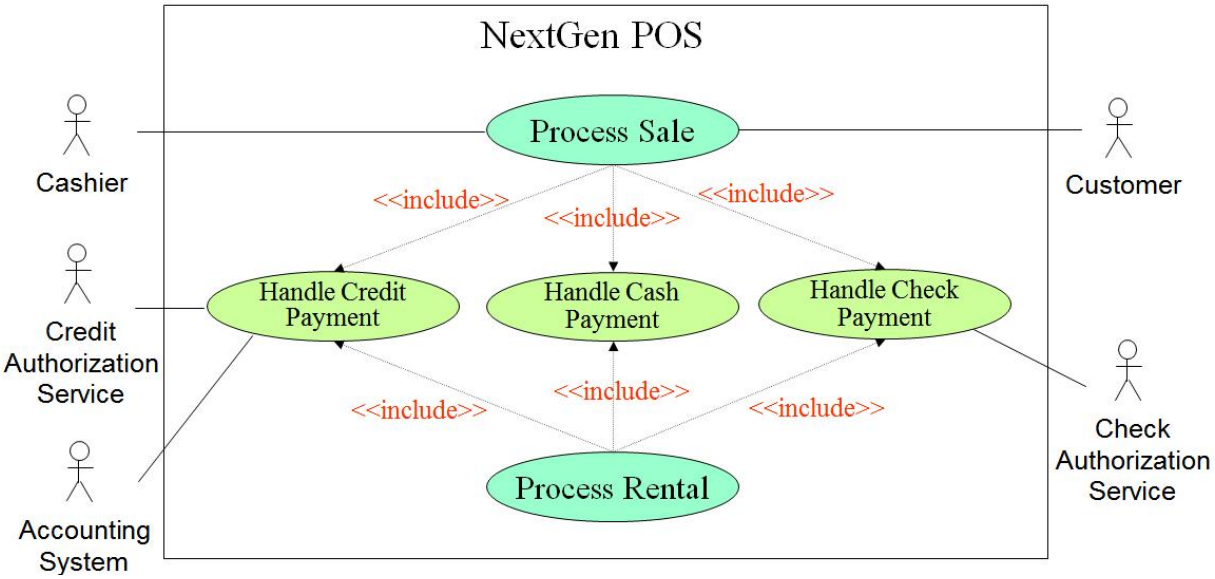
Use Case Diagrams



Organizing Use Cases

- Factoring out common behavior for reuse
  - <<include>> relationship
    - Formally, it was **<<use>>** relationship
- Adding optional sequence of events to the original use case
  - <<extend>> relationship

<<include>> Relationship



<<include>> Relationship (Cont'd)

UC1: Process Sale

…

Main Success Scenario:

1. Customer arrives at a POS checkout with goods to purchase.

…

7. Customer pays and System handles payment.

…

Extensions:

7b. Paying by credit: ***include Handle Credit Payment***.

7c. Paying by check: ***include Handle Check Payment***.

...

<<extend>> Relationship

UC5: Place Order

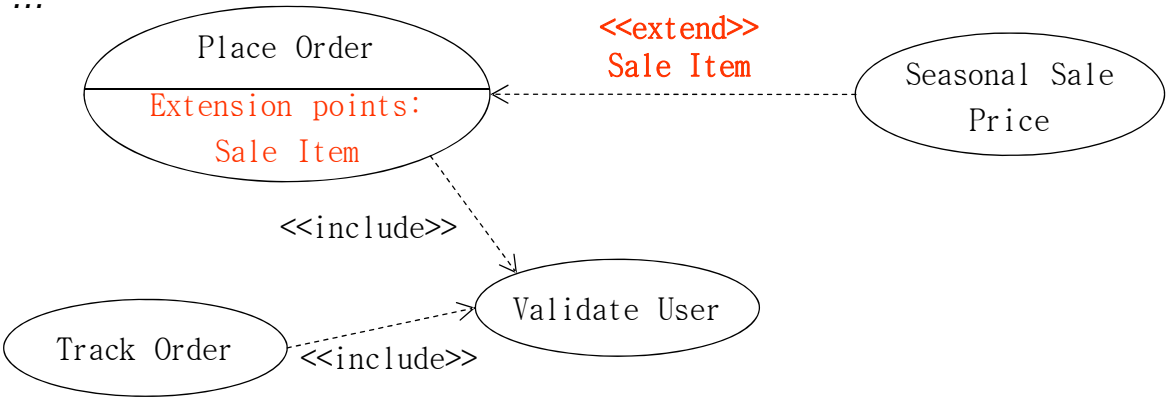
…

Extensions Points: *Sale Item*, step 2.

Main Success Scenario:

1. *include Validate User.*
2. *Collect the user's order items.*
3. *Submit the order for processing.*

…

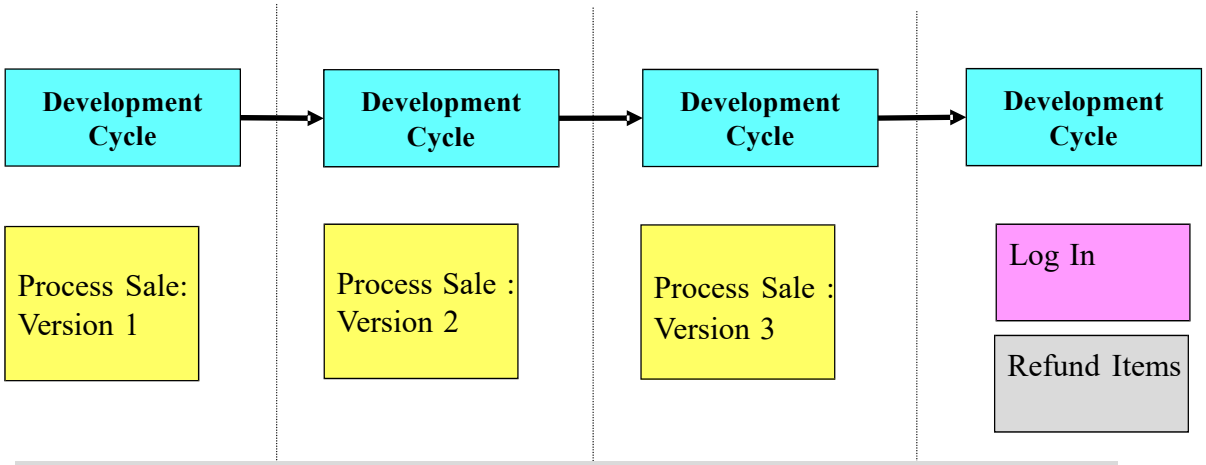


Use Cases
Within a Development Process

Iterative Development Cycle Phase Steps:

- Analysis phase
  - Write expanded essential use cases for those currently being tackled, if not already done.
- Design phase
  - Write real use cases for those currently being tackled, if not already done.

Use Cases Allocation
For Case Study



Process Sale - version 1 (cash payments, no inventory update, …)
Process Sale - version 2 (allow all payment types)
Process Sale - version 3 (complete version including inventory updates, …)

Use Cases - Why?

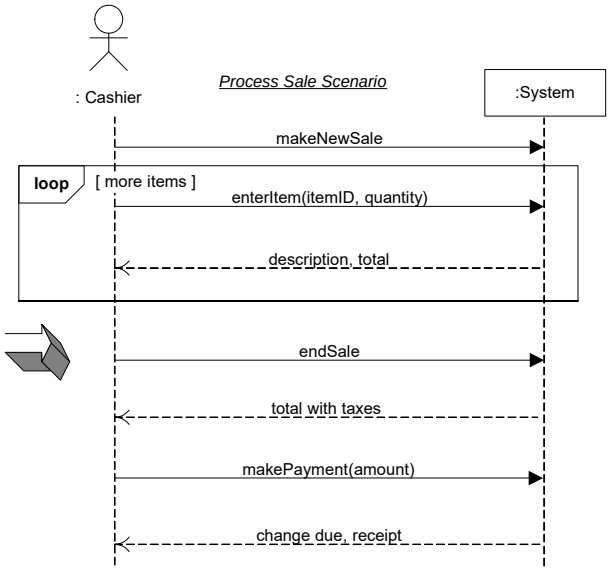
- To give concrete examples of what we are supposed to be implementing!
- A basis for an agreement between the clients and the developers
- A tool to make vague requirements more concrete
- A source for object analysis
- A source for the functionality of the system
- A unit for iterative & incremental software development
- The first version of the user's guide
- The basis of system testing

Object-Oriented Analysis and Design using UML and Patterns

System Sequence Diagram (SSD)

Simple cash-only *Process Sale* scenario:

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.
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.
...

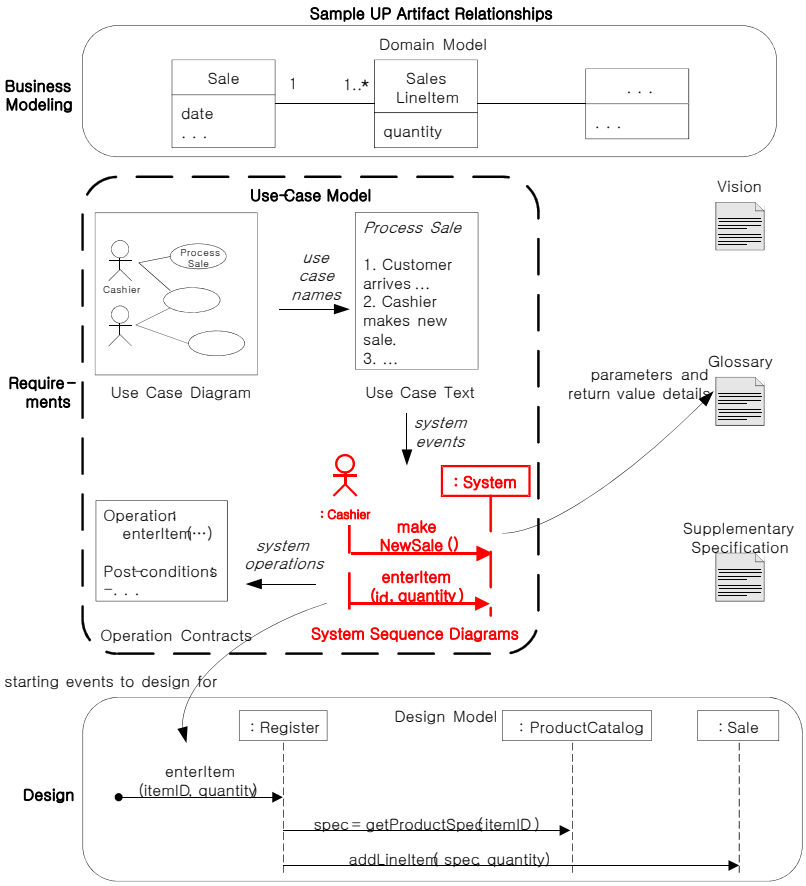


Use-Case Model:
Drawing System Sequence Diagrams

Objectives

- Identify system events and system operations.
- Create system sequence diagram for use cases.

Where are we?



Moving on Iteration 1

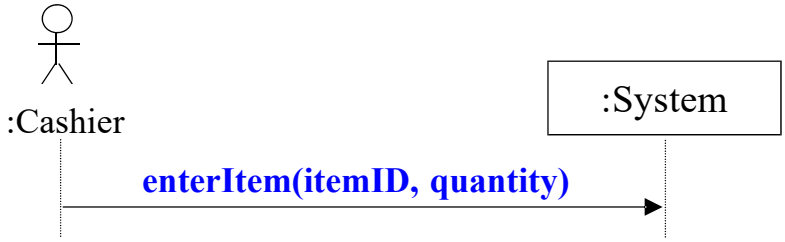
- **Cash-only success scenario *Process Sale*** (w/o remote collaborations), with the goal of starting a "wide and shallow" design and implementation that touches on many major architectural elements of the new system.

System Sequence Diagrams

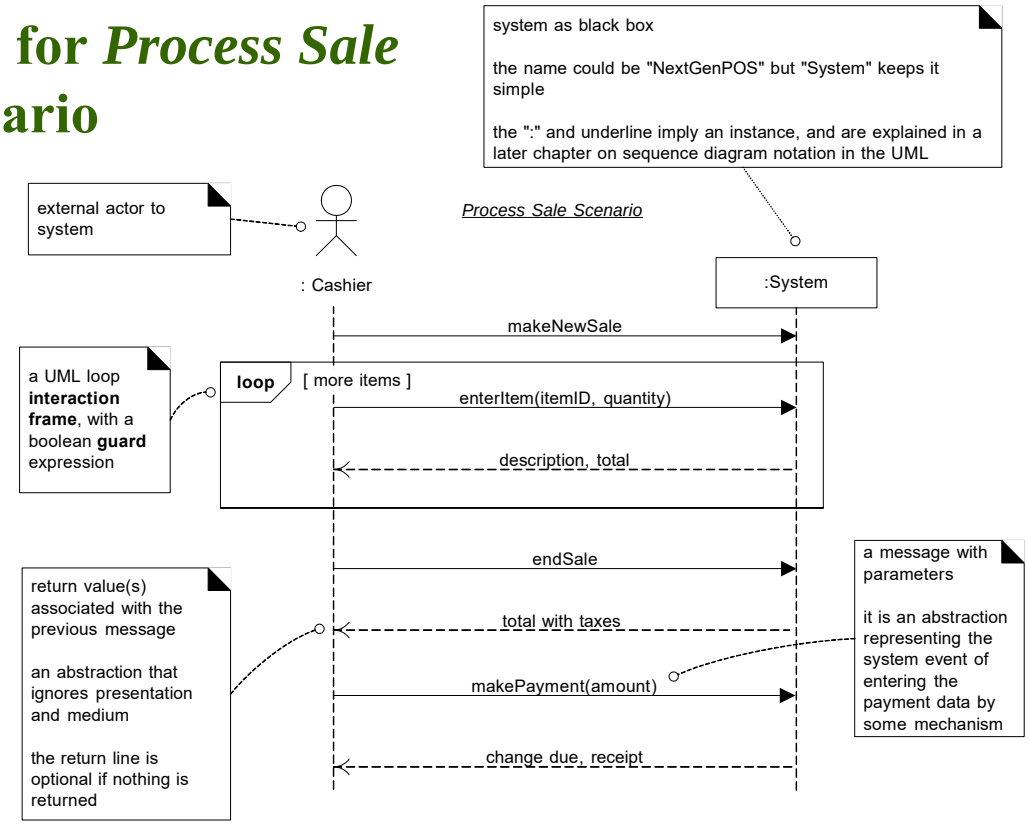
- System behavior is a description of *"what"* a system does, without explaining how it does it.
- A system sequence diagram captures the system behavior, for a particular scenario of a use case, by illustrating the events that external actors generate to SuD, their order, and inter-system events.
  - It allows us to treat the system as a black box.
- An SSD emphasizes those events that cross the system boundary from actors to systems.
- *An SSD should be done for the main success scenario of the use case, and frequent or complex alternative scenarios.*

System Events & System Operations

- A system event is an external event generated by an external actor to a system ===> *stimulus*.
- A system operation is an operation of the system that executes in response to a system event ===> *response*.
  - The set of all required system operations is determined by identifying the system events. The system events are derived from use cases.
- Both have the same name, such as **`enterItem`**.

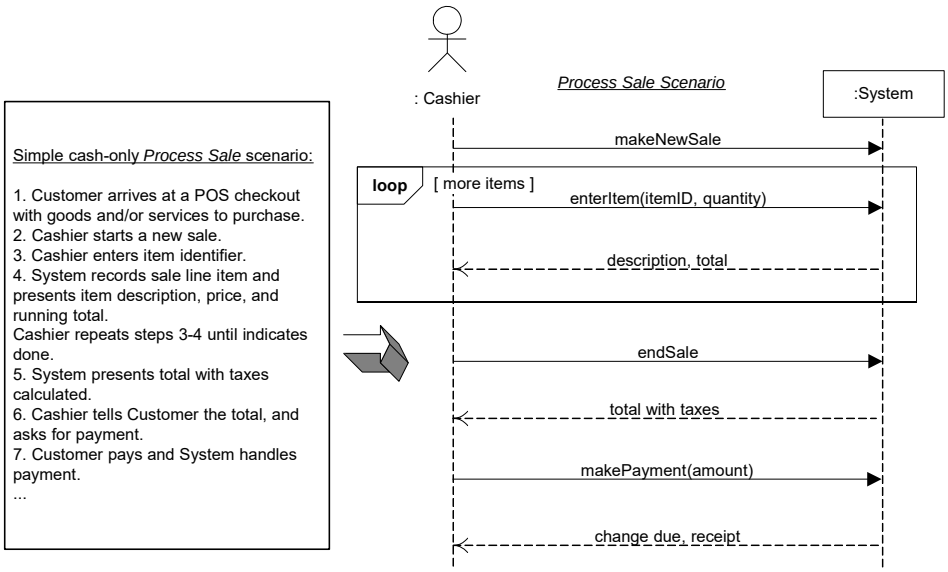


SSD for *Process Sale* scenario



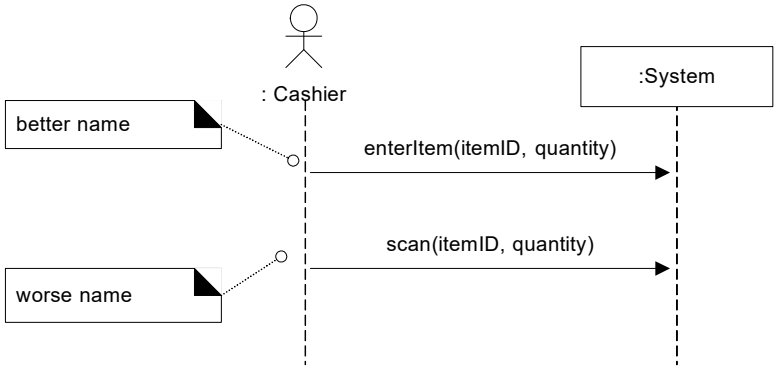
SSDs and Use Cases

- An SSD shows system events for a scenario of a use case, therefore it is generated from inspection of a use case.



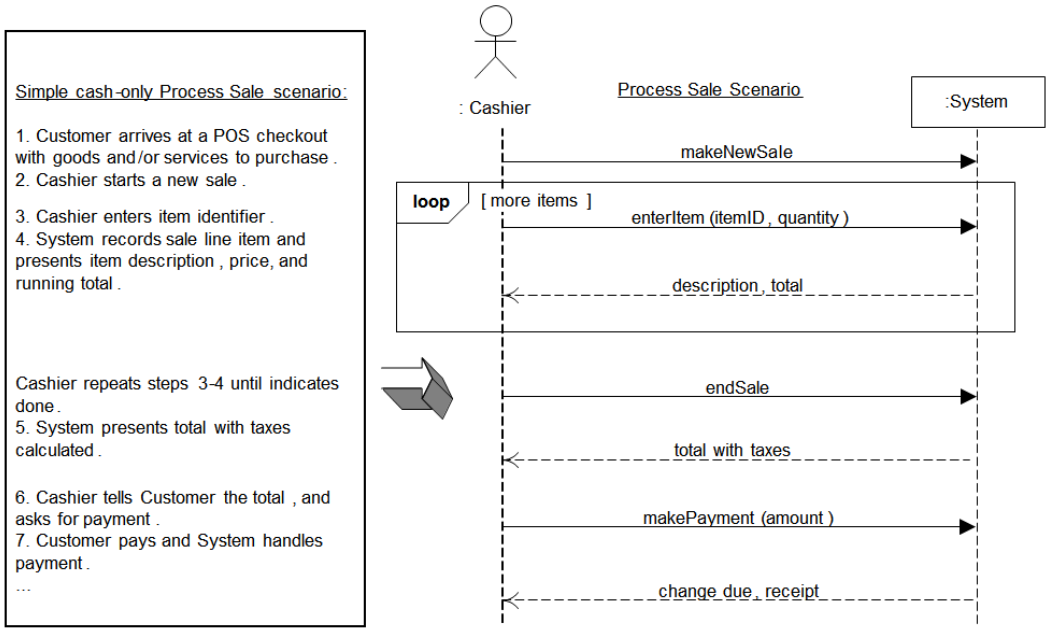
Naming System Events/Operations

- System events and their associated events should be *expressed at the (highest) level of intent* rather than in terms of physical input medium or interface widget.
- Usually start with a "verb" like *add…* , *enter…* , *make…* etc.



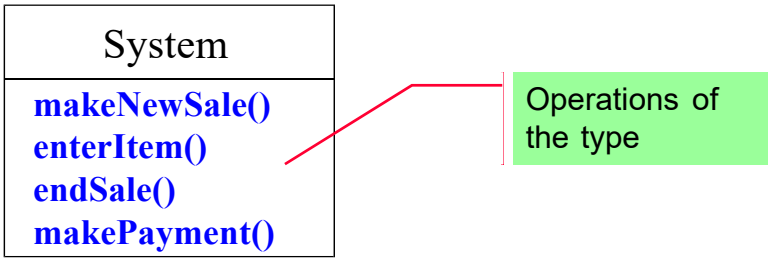
Showing Use Case Text

- The text provides the details and context; the diagram visually summarizes the interaction.



Recording System Operations

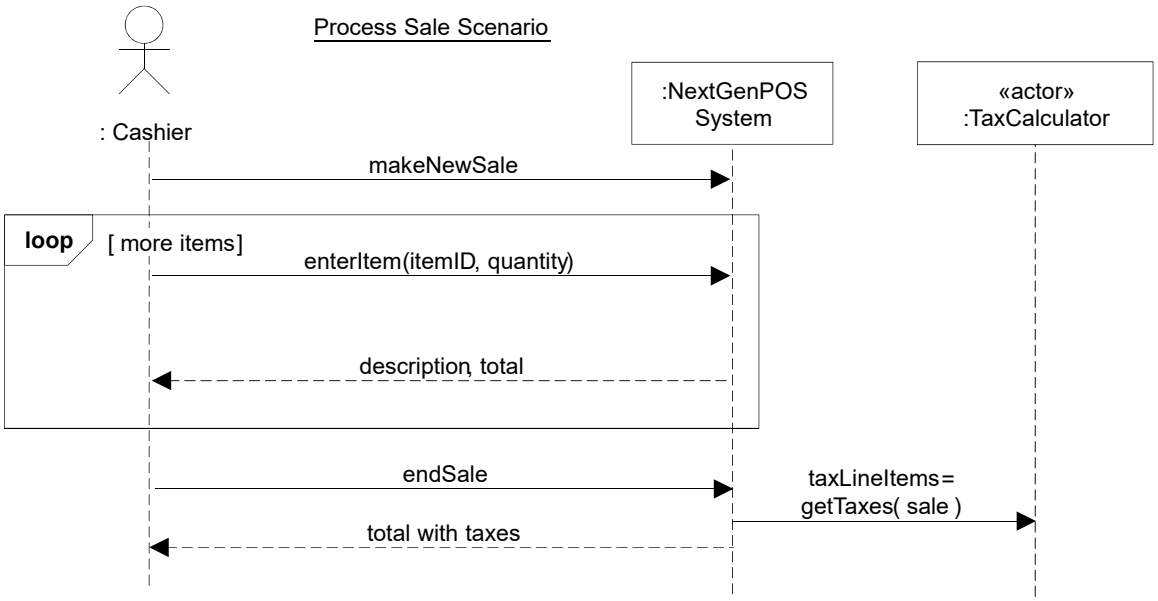
- UML class icon provides a compartment to record operations for a type.



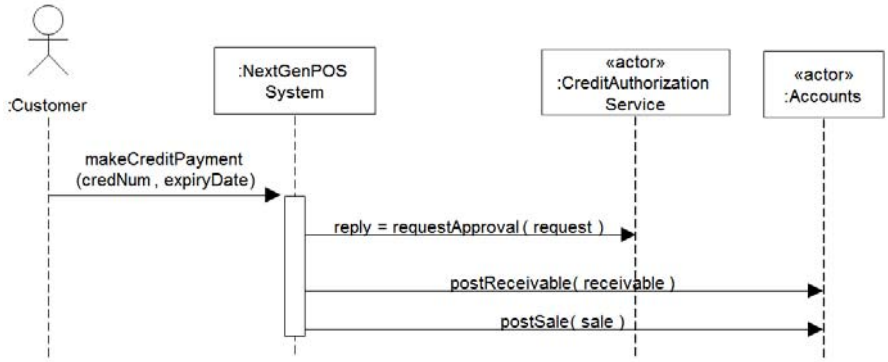
How To Make SSD

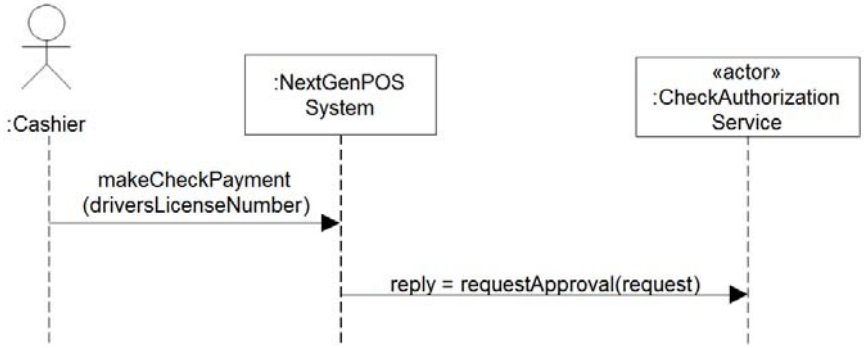
1. Draw a line representing the system as a black box.
2. Identify each actor that directly operates on the system. Draw a line for each such actor.
3. From the use case main success scenario text, identify the system (external) events that each actor generates. Illustrate them on the diagram.
4. Optionally, include the use case text to the left of the diagram.

More on SSD



More on SSD (Cont'd)





Exercises

Draw an SSD Derived from the following Main Success Scenario.

Use Case: Checkout books

Primary Actor: Librarian

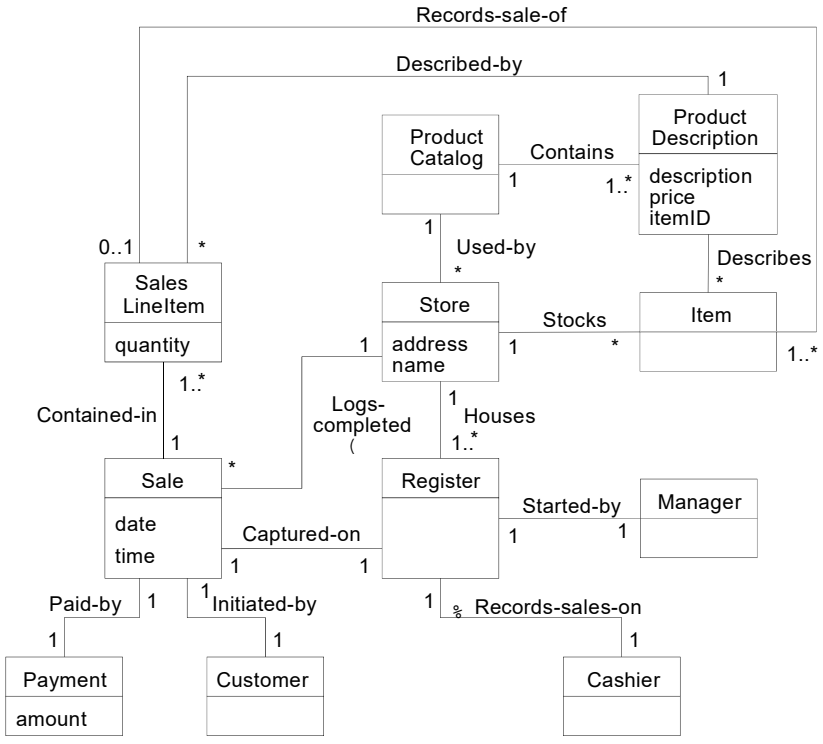
Main Success Scenario:

1. The Librarian enters the identity of a customer who wishes to check out books.
2. The System confirms that the customer is allowed to checkout books, and remembers the customer's identity.
3. The Librarian enters the identity of a book that the customer is checking out.
4. The System confirms that the book can circulate, calculates the due date based on whether the customer is a faculty member or a student, and records that the customer has checked out this book, which is due on the calculated due date.
5. The System tells the Librarian the due date.

The Librarian repeats steps 3-5 until indicates done.

Object-Oriented Analysis and Design using UML and Patterns

Domain Model & Contracts

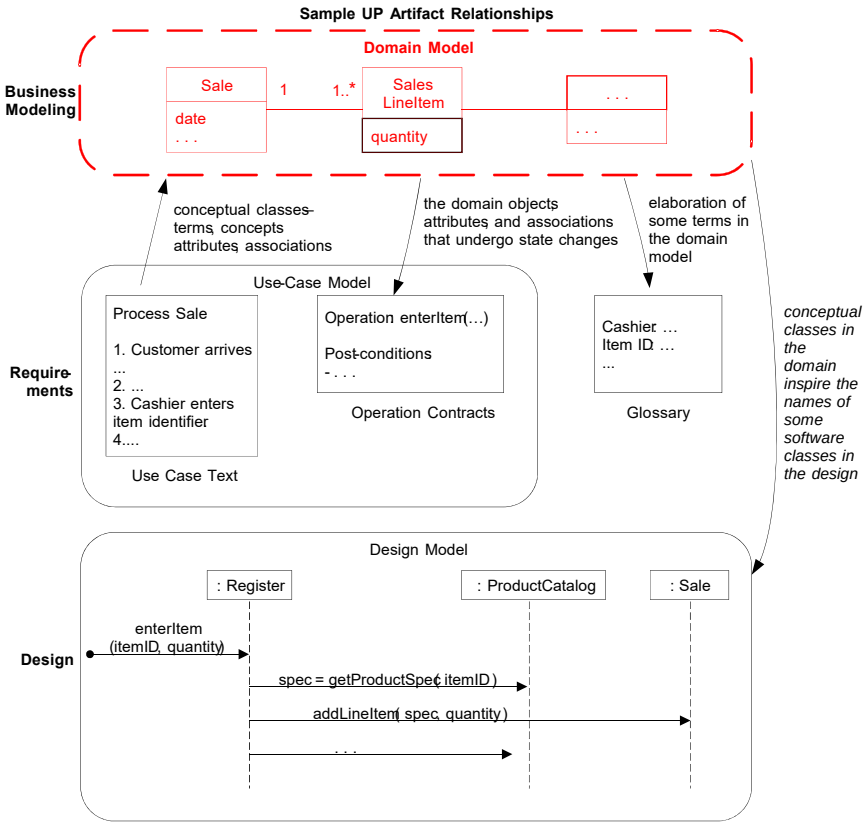


Domain Model: Visualizing Concepts

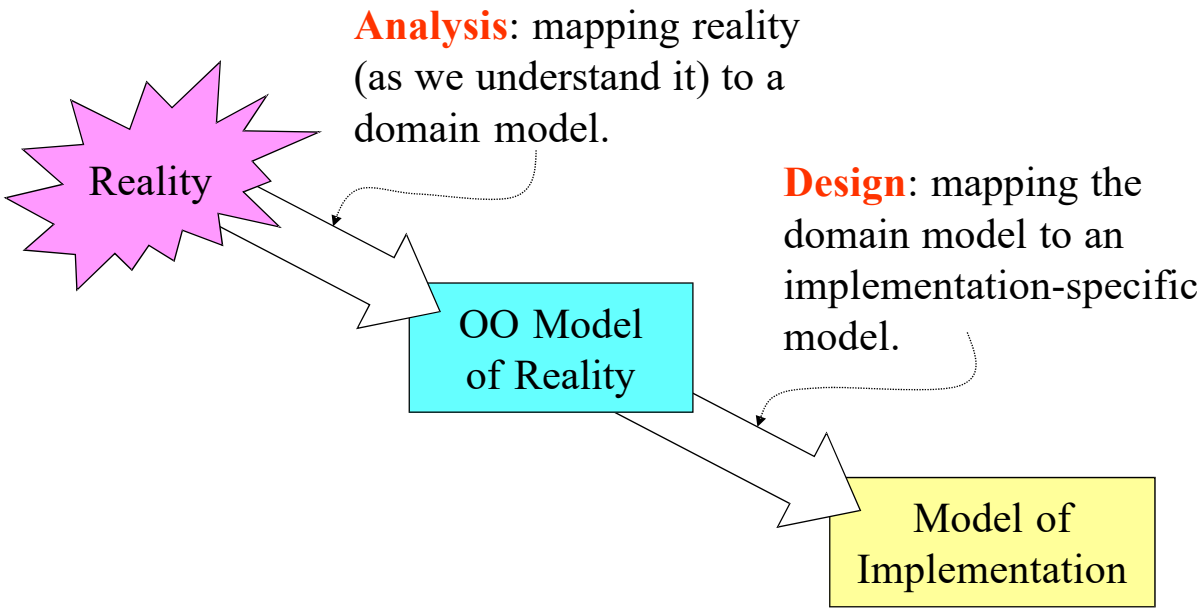
Objectives

- Identify conceptual classes related to the current iteration requirements.
- Create an initial domain model.
- Distinguish between correct and incorrect attributes.
- Add specification conceptual classes, when appropriate.
- Compare and contrast conceptual and implementation views.

Where are we?



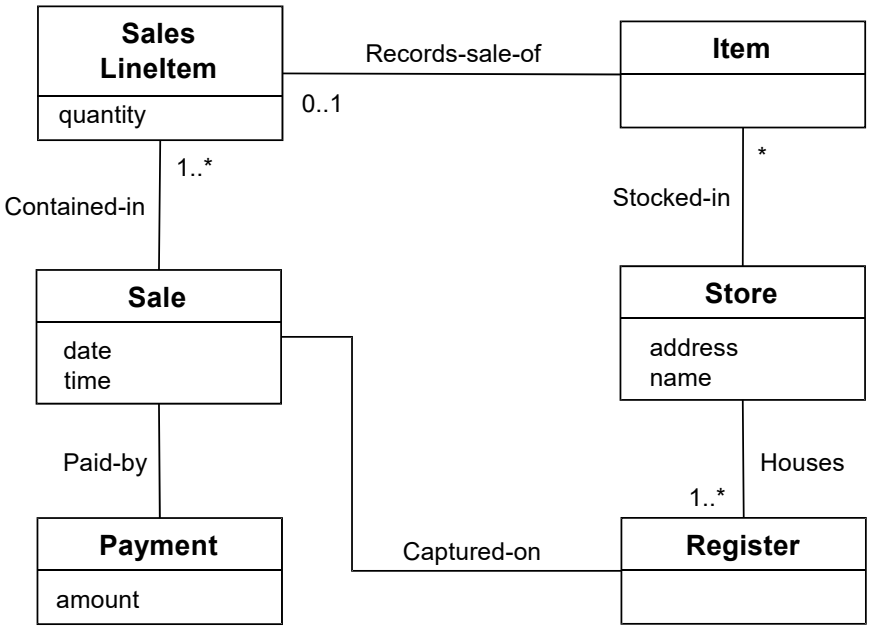
Analysis vs. Design



Domain Models

- A *domain model* is a *visual* representation of conceptual classes or real-world objects in the *problem* domain.
  - Also called *conceptual models*, *domain object models*, *analysis object models*.
  - A description of reality as we understand it, *not* of the designs (i.e., software classes or software objects with responsibilities).
  - The most important artifact to create during OO analysis.
- In UP, a domain model is represented by *class diagrams* in which no operations are defined. It may show:
  - **domain objects** or **conceptual classes**
  - relationships (mainly **associations**) between conceptual classes
  - **attributes** of conceptual classes

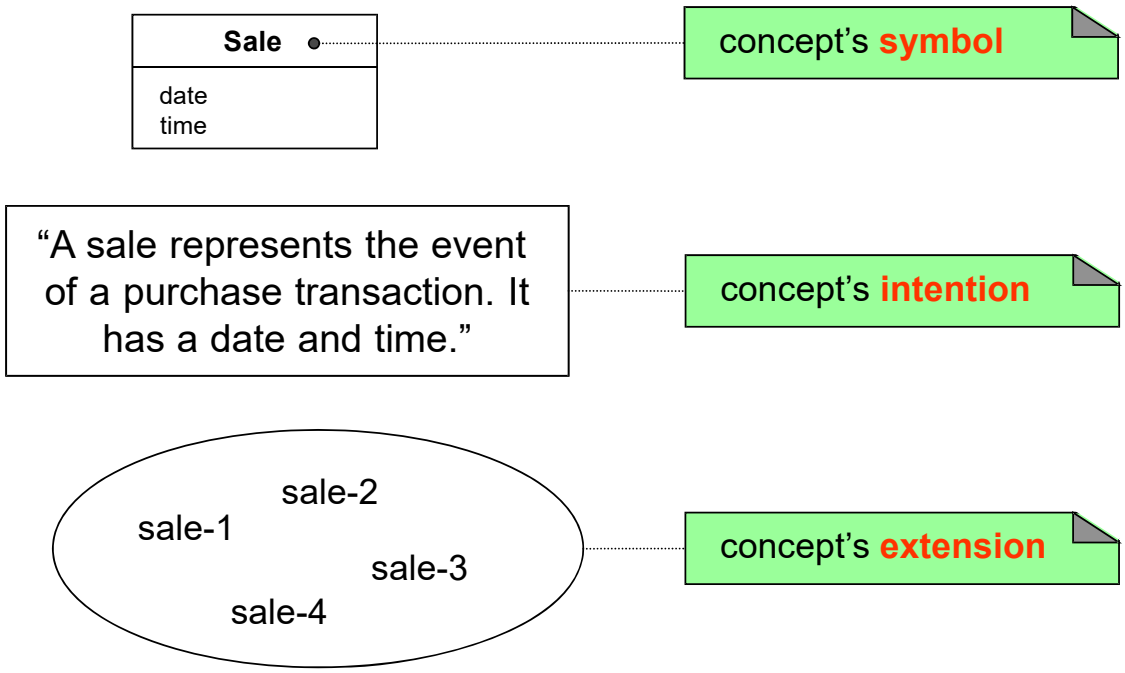
Example: Partial Domain Model



Conceptual Classes (or Key Abstractions)

- A **conceptual class** is an idea or notion that we apply to classify those things around us.
- More formally, a conceptual class may be considered in terms of
  - **Symbol**
    - words or images representing a conceptual class
  - **Intention**
    - the definition of a conceptual class
  - **Extension**
    - the set of examples to which the conceptual class applies

Example: A Conceptual Class for *Sale*



How to Identify Conceptual Classes

Two strategies to cover:

- Use a conceptual class category list.
- Identify noun phrases.

It is better to overspecify a domain model with lots of fine-grained conceptual classes, than to underspecify it.

Conceptual classes without attributes

Conceptual classes purely having behavioral role instead of an information role

List of Concept Categories

POST = Register

Physical objects (POST)	Organizations (Sales Dept.)
Designs, descriptions (Product Spec.)	Events (Sale, Robbery)
Places (Store)	Processes (Selling)
Transactions (Sale, Payment)	Polices (Refund policy)
People's roles (Cashier)	Catalogs (Product catalog)
Containers (Store, Bin)	Records/Contracts (Receipt)
Contained things (Item)	Financial Items (Credit line)
External systems (Credit bureau)	Manuals/Books (Employee manual)
Abstract concepts (Hunger)	

Finding Concepts with Noun Phrase Identification

Main Success Scenario (or Basic Flow)

1. **Customer** arrives at a **POST 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 2-3 until indicates done.

5. System presents total with **taxes** calculated.
6. Cahier tells Customer the total, and asks for **payment**.

…

9. System presents **receipt**.

…

Extensions (or Alternative Flows):

…

7a. Paying by cash:

1. Cahier enters the cash **amount tendered**.

…

Candidate Conceptual Classes for Sales Domain

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

- This list is constrained to the requirements and simplifications currently under construction – the simplified scenario of *Process Sale*.

Include Receipt in the Model?

Some factor to consider

- A receipt is a report of a sale. In general, showing a report of other information in a domain model is not useful since all its information is derived from other sources; it duplicates information found elsewhere. This is one reason to exclude it.
- A receipt has a special role in terms of the business rules: It usually confers the right to the bearer of the receipt to return bought items. This is a reason to show it in the model.

Include or Exclude?

- Since item returns are not being considered in this iteration, **Receipt** will be excluded. We may have a justification for its inclusion in later iterations (e.g., ***Handle Returns*** use case)

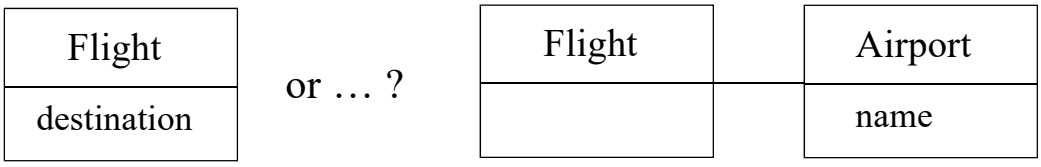
How to Make a DM?

1. List the candidate conceptual classes using CCCL and/or 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 (*will be discussed later soon*).
4. Add the attributes necessary to fulfill the information requirements (*also will be discussed later soon*).

Things To Remember

- Use the vocabulary of the problem domain when naming conceptual classes and attributes.
- Exclude irrelevant features.
- Do not confuse between conceptual classes and attributes.

If you don't think of some X as a number or text in the real world, X is probably a conceptual class, not an attribute.



If in doubt, make it a separate conceptual class!

Specification or Description Conceptual Classes

- An `Xspecification` or `XDescription` (e.g., `ProductDescription`) conceptual class records information (i.e., description) about `X`.
- When to Use?
  - There needs to be a description about item or service, independent of the current existence of any examples of those items or services.
  - Deleting instances of things they describe 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.

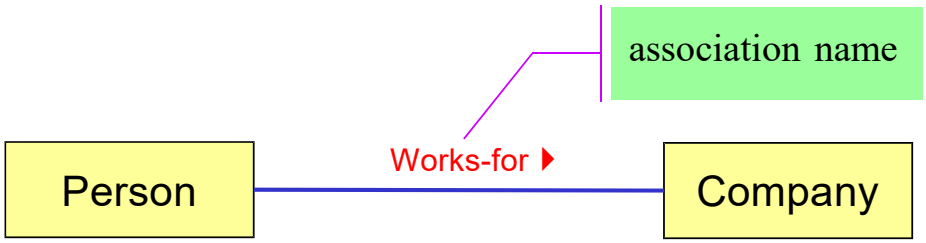
Domain Model - Adding Associations & Attributes

Objectives

- Identify associations for a domain model.
- Distinguish between need-to-know and comprehension-only associations.
- Identify attributes in a domain model.
- Distinguish between correct and incorrect attributes.

Association

- An association is a relationship between conceptual classes (or types) that indicates some meaningful and interesting connection.
- In the UML, an association is defined as a ***structural relationship*** between classes that specifies that objects of one class is connected to the objects of another class.



Useful Associations

Consider including the following associations in a domain model.

- 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.

Recommended Association Naming Conventions

- Use *TypeName-VerbPhrase-TypeName* format.
  - Where, the verb phrase creates a sequence that is readable and meaningful in the model context.
- Must start with a capital letter.
  - ***Stocks****(Store, Item),*
    Contains*(ProductCatalog, ProductSpecification)*
- A verb phrase should be constructed with hyphen.
  - ***Records-sale-of****(SalesLineItem, Item)*
- Make it read left-to-right or top-to-bottom.

Associations and Implementations

- During the analysis phase, 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 analytical sense.*
- When creating a conceptual model, we may define associations that are not necessary during construction. Conversely, we may discover associations that needed to be implemented but are missed during the analysis phase. In that case, the conceptual model should be updated to reflect these discoveries.
- In design phase, usually *an association is implemented as an attribute that points to an instance of the associated class.*

Useful Associations

- "Need-to-know" associations
  - Relationships that we need to remember to fulfill the requirements.
  - e.g., Records-sale-of(SalesLineItem, Item)
    ??? (Manager, Sale)
- Associations derived from *Common Associations List*.
- "Comprehension-only" associations
  - Relationships that aid in understanding of the important conceptual classes in the problem domain.
  - e.g., Initiated-by (Sale, Customer)

Common Associations

POST = Register

- Physical part (POST–Drawer)
- Logical part (LineItem–Sale)
- Physical containment (POST–Store)
- Logical containment (Description–Catalog)
- Description (ProductSpec–Item)
- Knows/Records (POST–Sale)
- Membership (Cashier–Store)
- Ownership (POST–Store)
- Subunit (Department–Store)
- Transaction (Payment–Customer, Payment–Sale)
- Communication (Customer–Cashier)

High Priority Associations

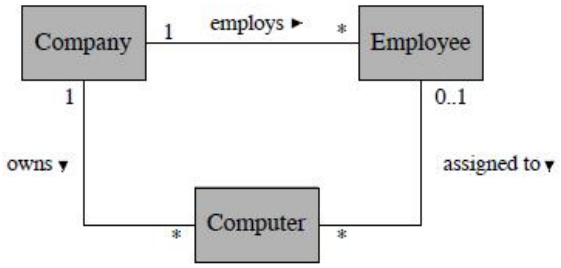
- A is a *physical* or *logical part* of B.
- A is *physically* or *logically contained* in/on B.
- A is *recorded* in B.

Summary of Associations

- An association is a (structural, usually permanent) relationship between conceptual classes that indicates some meaningful and interesting connection.
- Focus on those associations for which knowledge of the relationship needs to be preserved to satisfy the requirements ("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.

Summary of Associations
(Cont'd)

- Avoid showing redundant or derivable associations.

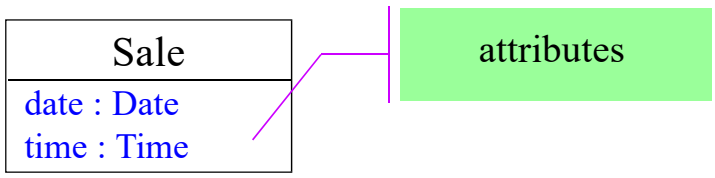


- Add choice comprehension-only associations to enrich critical understanding of the domain.

A conceptual model is not only an information model, but also a tool of communication!

Attributes

- An attribute is a logical data value of an object.



- In a domain model, an attribute should preferably be a pure data value (i.e., simple attribute), not a conceptual class (or type). Pure data values do not have an identity.

Common Simple Attributes

Very common simple attribute types include (primitive data types) :

- *Boolean*, *Date*, *Number*, *String (Text)*, *Time*

Other common simple types include (non-primitive data types):

- *Address*, *Color*, *Geometrics (Point, Rectangle, …)*, *Phone Number*, *Social Security Number*, *Universal Product Code (UPC)*, *SKU*, *ZIP or postal codes*, *enumerated types*.

Represent as non-primitive type for the following seemingly pure data values if …

- It is composed of separate sections
  - phone number, name of a 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 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 which is a generalization of various coding schemes such as UPC and EAN.

Represent as non-primitive type for the following seemingly pure data values if … (Cont'd)

OK

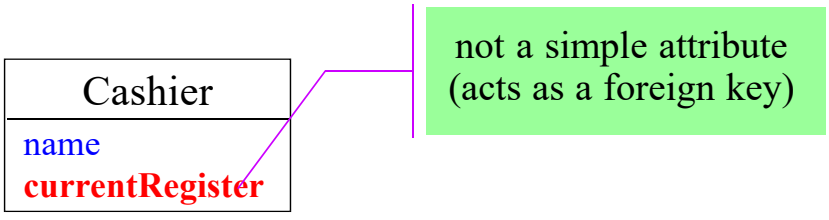
OK

A conceptual model is a tool of communication; choices about what is shown should be made with that consideration.

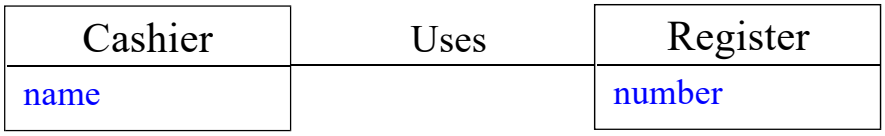
Wrong Attributes

- Sometime, associations are mistakenly represented as attributes.

Worse



Better



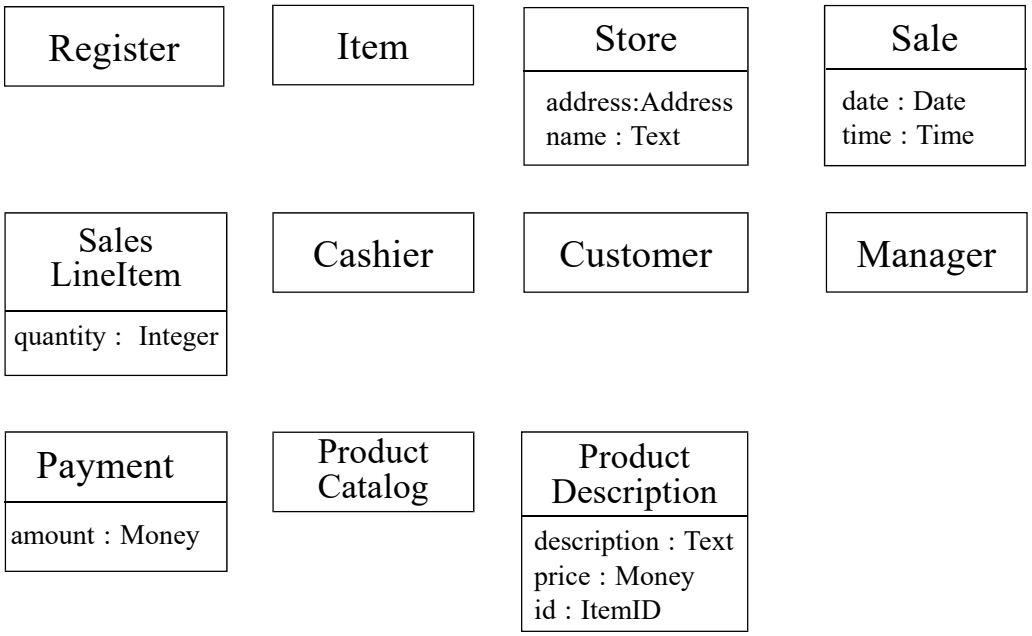
When in doubt, define something as a separate conceptual Class rather than as an attribute.

Attributes for NextGen System

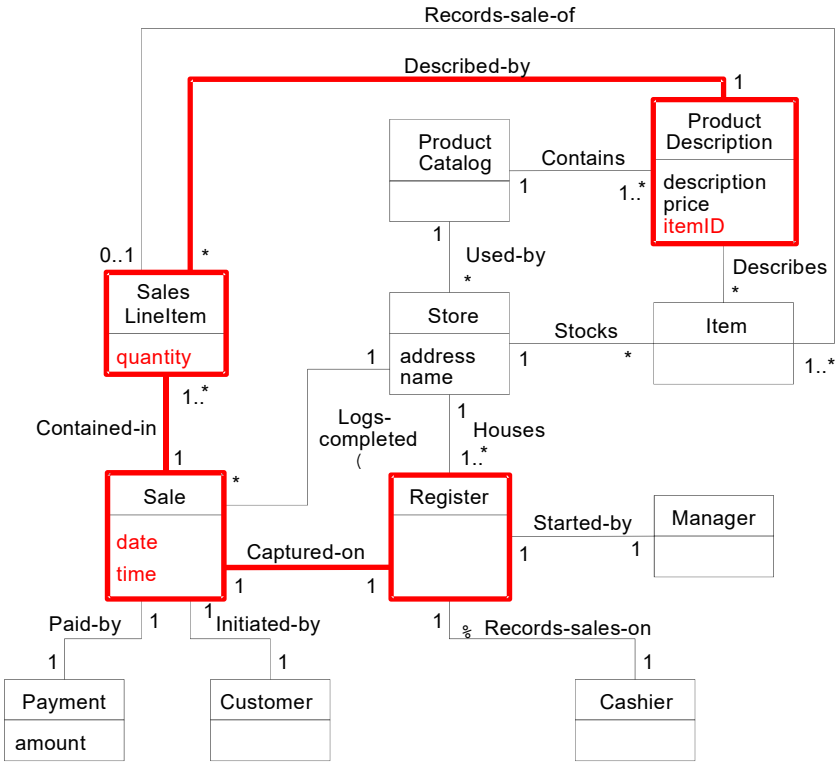
- A list of attributes need to be generated by consulting:
  - requirement specification
  - current use case under consideration
  - simplification, clarification, and assumption documents
- Some attributes may not be identified during analysis. This is acceptable; during design and implementation phases the remaining attributes will be discovered and added.

Attributes in the NextGen Model

(Reflecting *Process Sale* Scenarios from this Iteration)



A Partial Domain Model

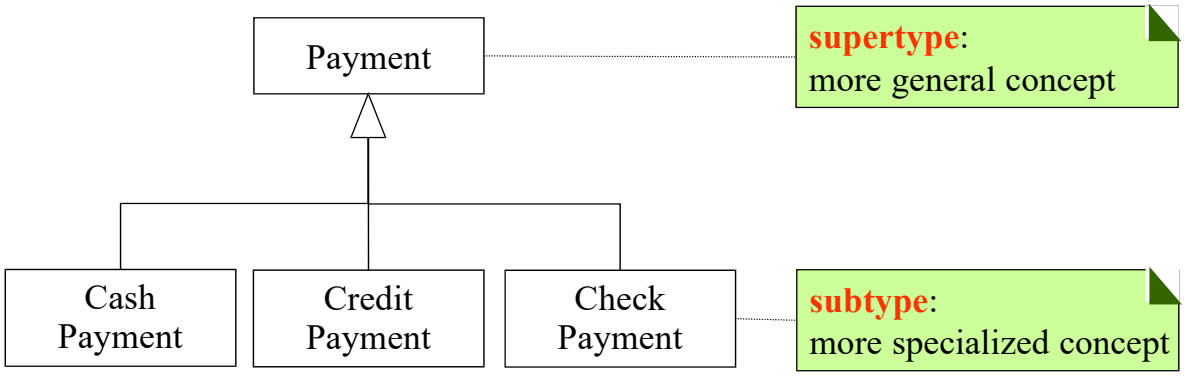


Polishing Domain Model

- Add generalization/specialization relationships.
- Add aggregation and composition aggregation relationships.
- Add association types.
- Add self associations.
- Add qualified associations.
- Etc.

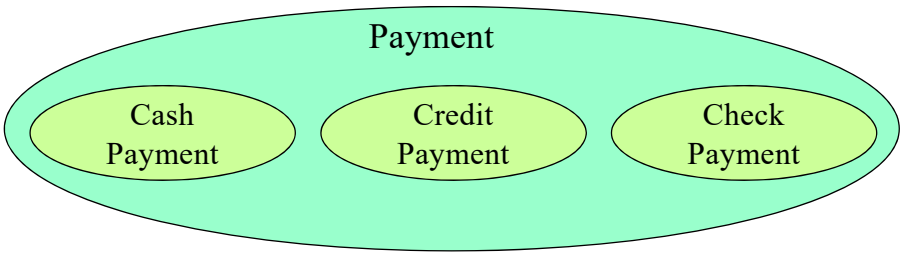
Generalization/Specialization

- A group of similar concepts (or types) can be organized into generalization/specialization type hierarchy.
- *Generalization* refers to identifying commonality among concepts and defining *supertype* and *subtype* relationships.



Supertype and Subtype

- All members of a subtype set are members of their super type set.
- Everything that applies to the supertype must also be applied to the subtype ==> *subtype "is a" supertype (100% rule).*
- *Corollary: Whenever an object of supertype is expected, an object of the subtype can also be used, but not vice versa ==> Liskov Substitution Principle (LSP).*

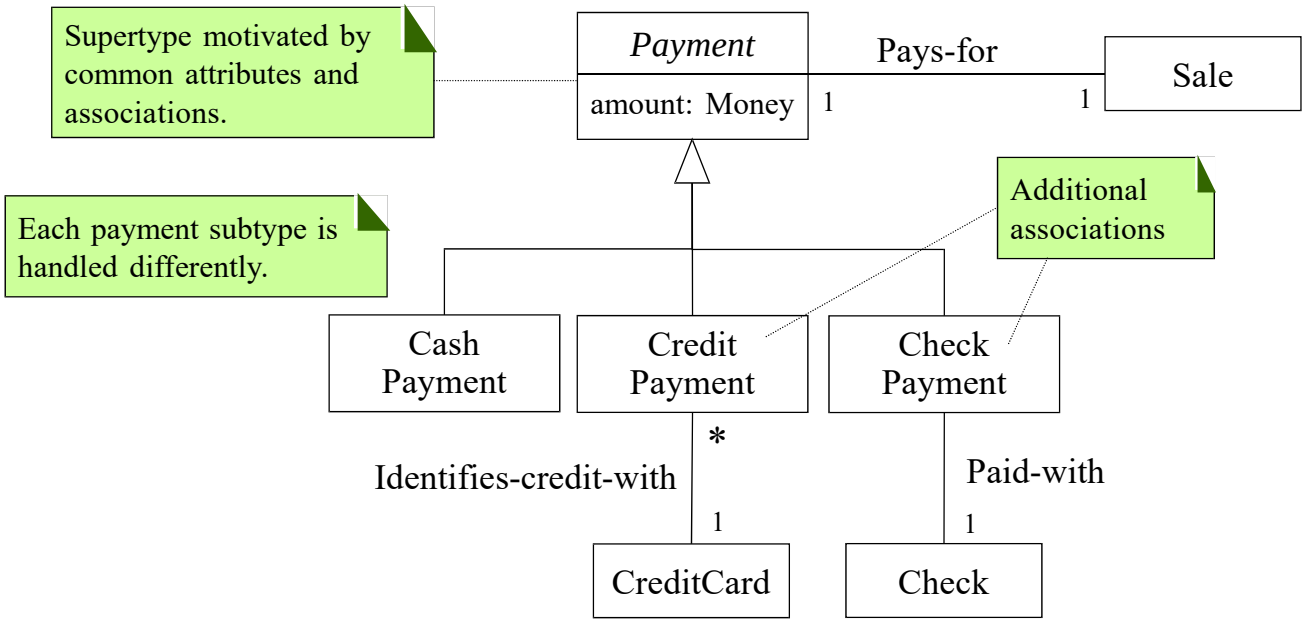


When to Define a Type Hierarchy

- The subtype has additional attributes of interest.
- The subtype has additional associations of interest.
- The subtype is operated on, handled, reacted to, or manipulated differently than or other subtypes, in ways that are of interest.
- The subtype represents an animated thing (for example, animal, robot) that behaves differently than the supertype or other subtypes, in ways that are of interest.
  - Overriding

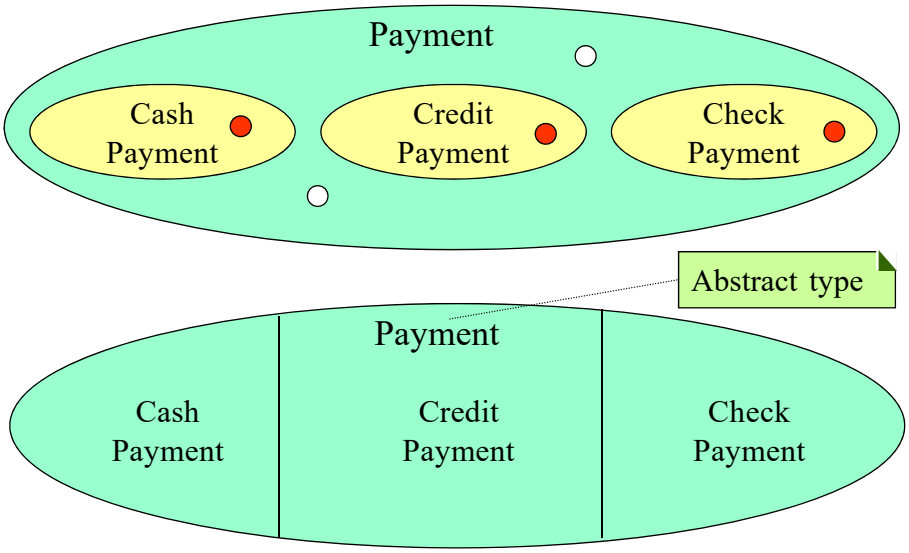
At all times, check whether the "is a" relationship (or 100% rule) is preserved.

Payment Hierarchy



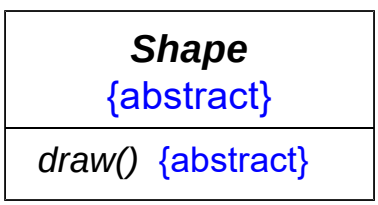
Abstract Types

- If every member of a type *T* must also be a member of a subtype, then type *T* is called an *abstract type*.



Abstract Class/Methods

- If an abstract type is implemented in software as a class during design phase, it will usually be represented by an *abstract class*, meaning no instance may be created for the class.
- An *abstract method* is one that is declared in an abstract class, but not implemented.



Abstract class

Modeling Changing State

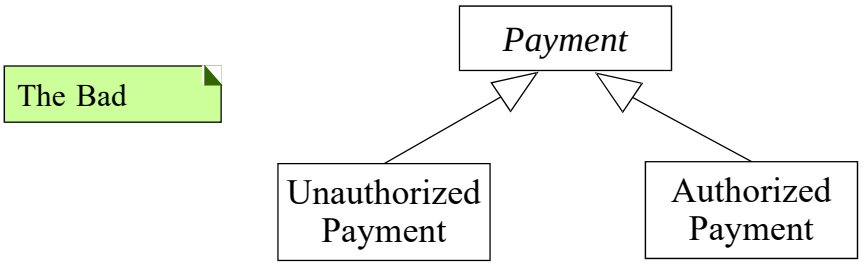
- Do not model the states of a concept *X* as subtypes of *X*. Rather:

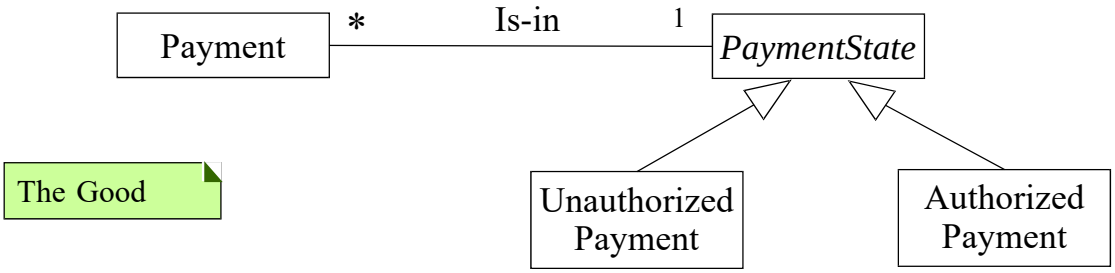
1. Use *State Pattern*
   - Define a state hierarchy and associate the states with *X*

   or

2. Ignore showing the states of a concept in the domain model; show the states in state diagrams instead.

The Bad and The Good





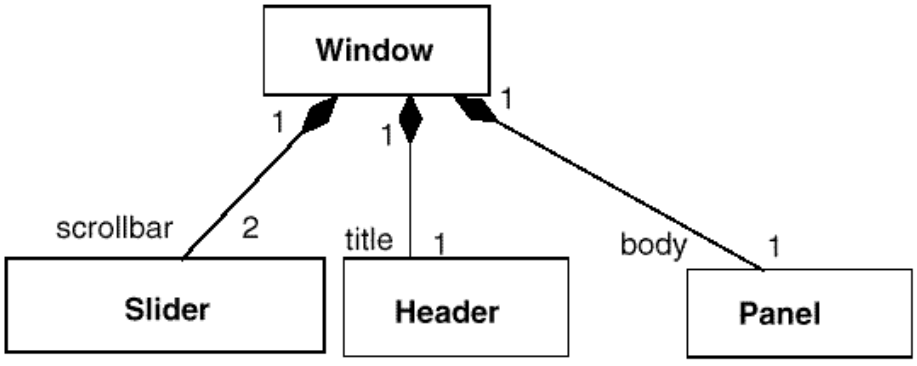
Aggregation (Review)

- No semantic difference from Association.
- Aggregation represents a ***whole/part*** relationship, i.e, the composite object (whole) is made up of other objects (parts).


```
class Window {
public: ...
  void  addShape(Shape*); Shape* removeShape(Shape*);
private:
  vector<Shape*> itsShapes;
};
```


Composition (Review)

- A hard form of aggregation denoting ***ownership***.
- Composites control the lifetime of their constituents.
  - Ownership can can*not* be shared.

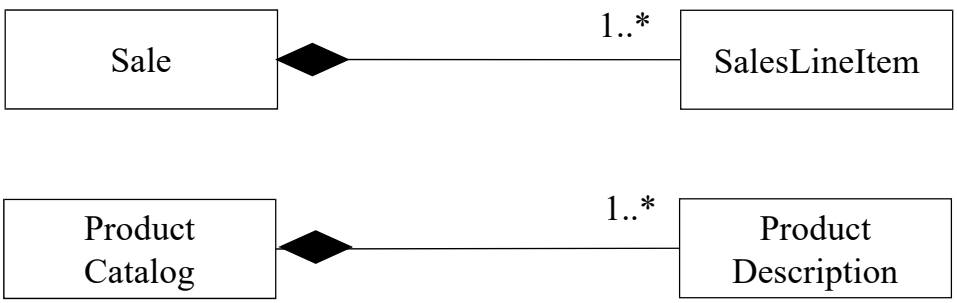


When to Show Aggregation

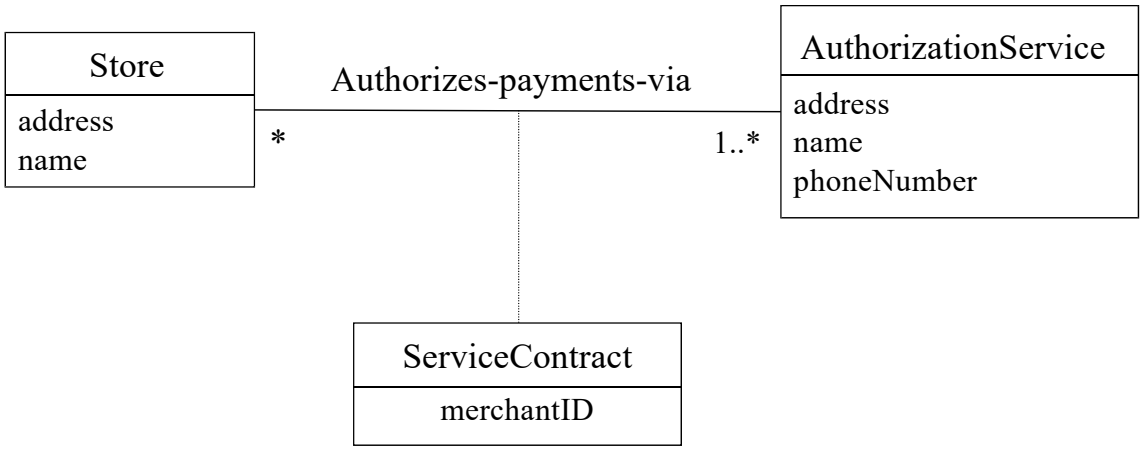
- The lifetime of the part is bound within the lifetime of the composite – there is a create-delete dependency of the part on the whole.
- There is an obvious whole-part physical or logical assembly.
- Some properties of the composite propagate to the parts, such as its location.
- Operations applied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

If in doubt, leave it out!

Aggregation in NextGen POS

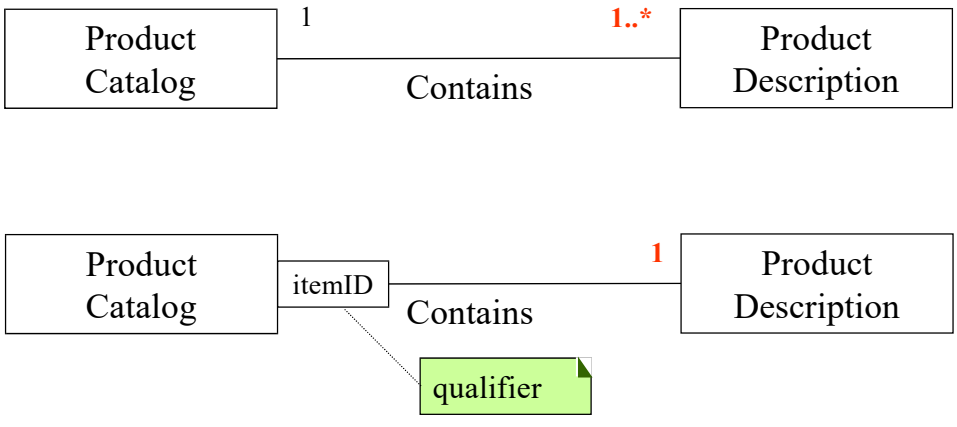


Associative Type at NextGen POS



Qualified Associations

- A *qualifier* 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*.



Summary of Domain Models

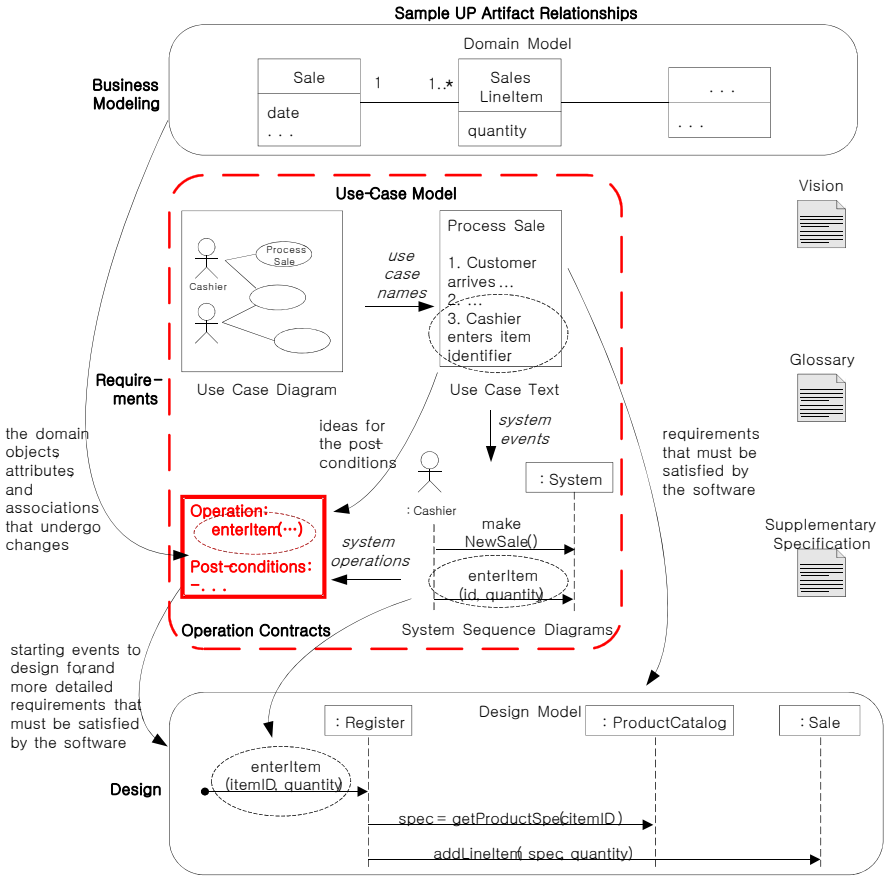
- There is no such thing as a single correct model.
- All models are approximations of the domain that we are attempting to model.
- A good domain model captures the essential abstractions and information required to understand the domain in the context of the current requirements, and aids people in understanding the domain --- its concepts, terminology, and relationships.

Use-Case Model: Adding Detail With Operation Contracts

Objectives

- Create contracts for system operations.

Where are we?



Specification by Contracts

- **Contract**: explicit agreement that binds a provider and a client.
  - States the rules that restrict use of a service (operation) by the client -- **pre-conditions**
  - States the guaranteed result of legally invoking the operations -- **post-conditions**
- The notion of contracts is due to Bertrand Meyer.
  - Good reference: B. Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall PTR, 1997.

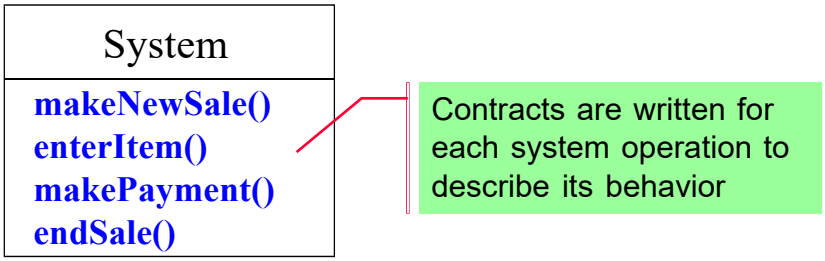
Pre-
Post-

Contracts in Software

	Obligations	Benefits
Client	*(Must ensure precondition)* Make sure table is not full and key is a non-empty string.	*(May benefit from postcondition)* Get updated table where the given element now appears, associated with the given key.
Supplier	*(Must ensure postcondition)* Record given element in table, associated with given key.	*(May assume precondition)* No need to do anything if table is full, or key is empty string.

System Behavior and Contracts

- System contracts describe detailed system behavior in term of state changes to objects in the Domain Model, after a system operation has executed.
- A contract can be written for an individual operation of a class (and interface), or for a system operation.
- Therefore, system operation contracts are associated with system operations like makeNewSale(), enterItem(), makePayment(), endSale(), etc.

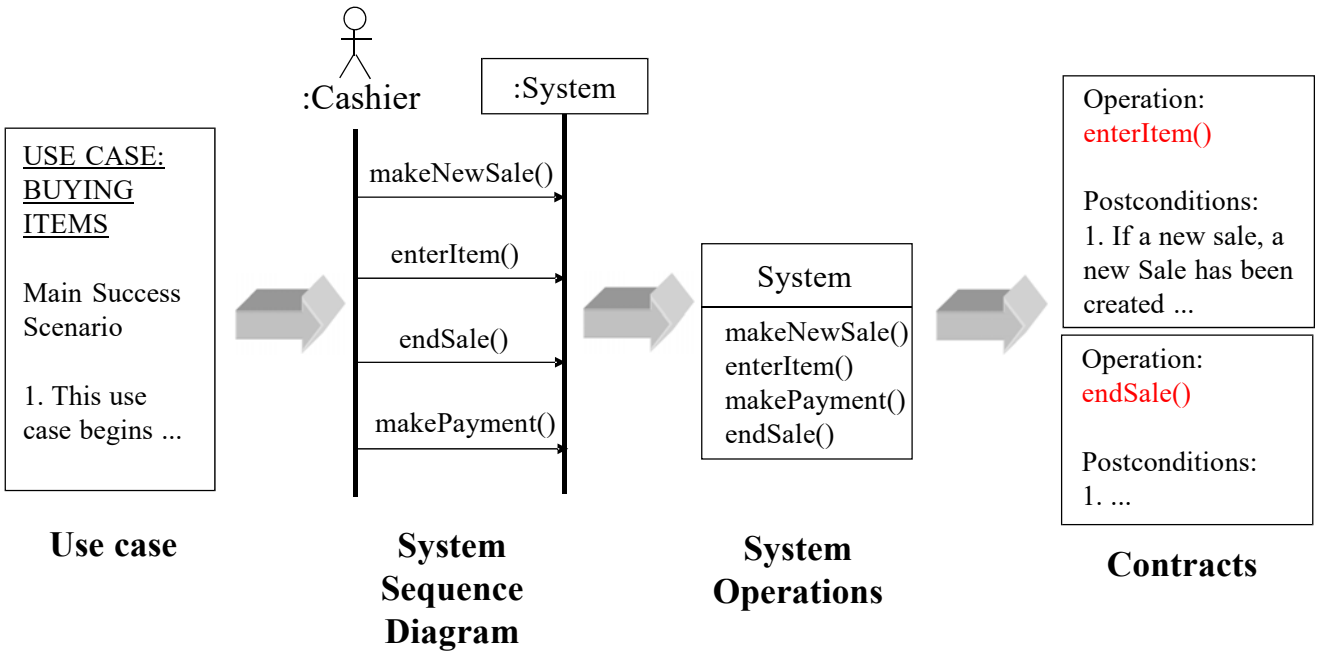


How to Make A Contract

To make contract for each use case:

1. Identify the system operations from the system sequence diagrams.
2. For system operations that are complex and perhaps subtle in their results, or which are not clear in the use case, construct a contract.
3. To describe the postconditions, use the following categories:
   - Instance creation and deletion
   - Attribute modification
   - Associations formed and broken.

Contracts and Other Artifacts



Contract CO1: makeNewSale

Operation: makeNewSale()
Cross References: Use Cases: Process Sale
Pre-conditions: none
Post-conditions:

- A ***Sale*** instance ***s*** was created (instance creation).
- ***s*** was associated with the ***Register*** (association formed).
- Attributes of ***s*** were initialized.

Contract CO2: enterItem

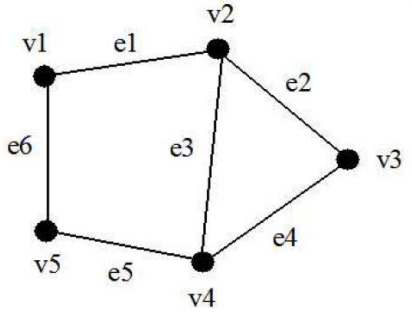
Operation: enterItem(itemID : ItemID, quantity : integer)
Cross References: Use Cases: Process Sale
Pre-conditions: There is a sale underway.
Post-conditions:

- A ***SalesLineItem*** instance ***sli*** was created (instance creation)
- ***sli*** was associated with the current ***Sale*** (association formed).
- ***sli.quantity*** become quantity (attribute modification).
- ***sli*** was associated with a ***ProductSpecification***, based on ***itemID*** match (association formed).

Exercises

문제. 다음에 주어진 설명을 잘 읽고, 언급된 conceptual class들 사이의 각종 관계(association, aggregation, composition aggregation, generalization/specialization), role names, multiplicity 등을 가능한 한 상세히 보여주는 UML class diagram을 그리시오.

(1) 비방향성 그래프(undirected graph)를 위한 UML class diagram을 보이시오. 비방향성 그래프(undirected graph)는 vertex의 집합과 edge의 집합으로 구성된다. 하나의 edge는 두 개의 vertex를 연결한다. 비방향성 그래프를 나타내는 클래스 다이어그램은 그래프의 구조 즉, 연결성을 정확히 표시해야 하며, 각 vertex의 위치나 edge의 길이 따위와 같은 기하학적인 세부사항은 고려하지 않는다. 다음 그림은 비방향성 그래프의 한 가지 예를 보여주고 있다.



(2) 아래에 주어진 conceptual class들 만을 사용하여 UML Class Diagram을 완성하시오.

University	Person	Professor	Student
Department	PhD Student	Master Student	

- University consists of 1 or more Departments.
- Each Person is associated with a Department, and each Department consists of 1 or more Persons.
- Person is either a Professor or a Student.
- Student is either a PhD Student or a Master's Student.
- For each Department, a Professor serves as its chair. Moreover, a Professor serves as the chair of at most one Department.
- Professor has one or more Students as a teaching assistant. Each Student assists at most one Professor.
- Professor may serve as the thesis advisor of at most 10 PhD Students. Moreover, a PhD student has at most 2 thesis advisors.

(3) 그룹핑(grouping) 기술은 다양한 그래픽 편집기에서 사용되고 있다. 그룹핑을 지원하는 그래픽 문서 편집기(graphic document editor)를 위한 UML class diagram을 보이시오.

"문서(document)는 여러 개의 시트(sheet)로 구성되며, 각 시트는 텍스트(text), 기하 객체(geometrical objects), 그룹(group)과 같은 그림 객체(drawing objects)들을 포함한다. 하나의 그룹은 단순히 그림 객체들의 집합으로서 다른 그룹을 원소로서 가질 수 있다. 하나의 그룹은 반드시 적어도 두 개 이상의 그림 객체로 구성되어야 한다. 하나의 그림 객체는 동시에 두 개 이상의 그룹 멤버가 될 수 없다. 기하 객체는 원(circles), 타원(ellipses), 직사각형(rectangles), 선(lines), 정사각형(squares) 등을 의미한다."