Object-Oriented Analysis and Design

PART1: ANALYSIS

Textbook

Text: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Craig Larman, ISBN: 013 148 9062, Prentice-Hall, 2005

What will we learn?

- We will learn the skills needed for good object-oriented analysis and design
- We will utilize Unified Modeling Language (UML)
  - Be careful: Just knowing how to draw UML diagrams or create models does not imply good OOA/OOD!
  - UML is used mostly in this course as a diagraming method, a common notation
- Responsibility-Driven Design
  - How should responsibilities be assigned to classes of objects?
  - How should objects collaborate?
  - What classes should do what?
- For many common problems, these questions have been answered
  - Use existing best-practices, or *patterns*

What will we learn?

- We will learn how to apply OOA/OOD to several case studies, which will be referred to throughout the course
- We will learn how to do proper *requirements analysis*, and write *use cases*.
  - The first step in most projects, and the most important!
  - Not just for understanding the problem, but also to ensure common terminology, etc.
- What is the development process? How does OOA/OOD fit in?
  - We will consider the *agile* development process, as part of the Unified Process (UP). This is an iterative development process that is very popular today
  - The OOA/OOD concepts we will learn can be applied to other development processes as well
- We will also learn fundamental principles in object design and responsibility assignment: GRASP (General Responsibility Assignment Software Patterns)

We will:

- Apply principles and patterns to create better object designs
- Iteratively follow a set of common activities in analysis and design, based on the agile approach to the UP as an example
- Create frequently used diagrams in the UML notation
- Be able to skillfully assign responsibilities to software objects
  - Identify the objects that make up the system, or domain
  - Assign responsibilities to them – what do they do, how do they interact?
- Apply the GRASP principles in OOA/OOD

Objects

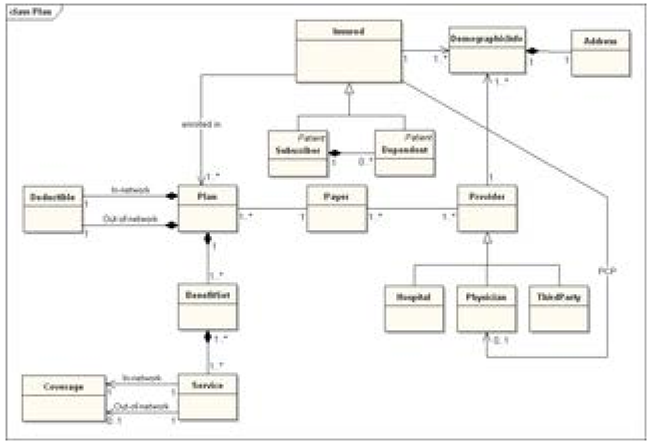
From Merriam-Webster:

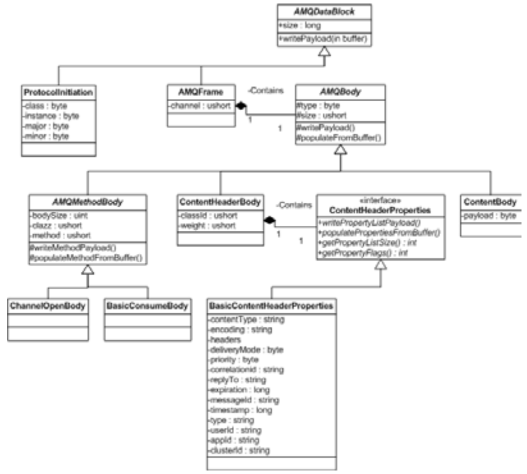
- “something material that may be perceived by the senses”

Look around this room, and imagine having to explain to someone who has never taken a class what happens here …

- You would explain the activity that occurs, and you would identify specific objects that play a role in that activity (Chairs, tables, projectors, students, professor, white board, etc.) to someone who has never seen these things …
- Each of these objects is well defined, and plays a separate role in the story. There may be multiple copies of chairs, but a chair is very different from a projector – they have different *responsibilities*
- You would not describe the action by saying “The classroom allows students to sit, and the classroom allows the professor to display slides, … “ etc. This would make the “classroom” too complex – almost magical
- You would define the various objects in this domain, and use them to tell the story and describe the action

OOA/OOD





Analyze the system

Model the system

Design the software

Analysis and Design:

- Analysis is the *investigation* of the problem - **what** are we trying to do?
  - Here is where use cases are created and requirements analysis are done
- Design is a *conceptual solution* that meets the requirements – **how** can we solve the problem
  - Note: Design is *not* implementation
  - UML diagrams are not code (although some modeling software does allow code generation)
- Object-oriented analysis: Investigate the problem, identify and describe the objects (or concepts) in the problem domain
  - Also, define the domain!
- Object-oriented design: Considering the results of the analysis, define the software classes and how they relate to each other
- Not every object in the problem domain corresponds to a class in the design model, and viceversa
- Where do we assign responsibilities to the objects? Probably a little in both parts

UML

“The Unified Modeling Language is a visual language for specifying, constructing, and documenting the artifacts of systems.” - *OMG, 2003*

Standard for diagramming notation

We will use UML to sketch out our systems

UML can be used (by modeling packages) to auto-generate code directly from the model diagrams

Different perspectives:

- Conceptual Perspective – defining the problem domain: Raw class diagrams, maybe mention some attributes (Domain Model)
- Specification Perspective – defining the software classes: Design Class diagram, which shows the actual software classes and their methods, attributes

UML

We will explore the details of UML diagramming

For now, understand that UML is a language – it is used to communicate information

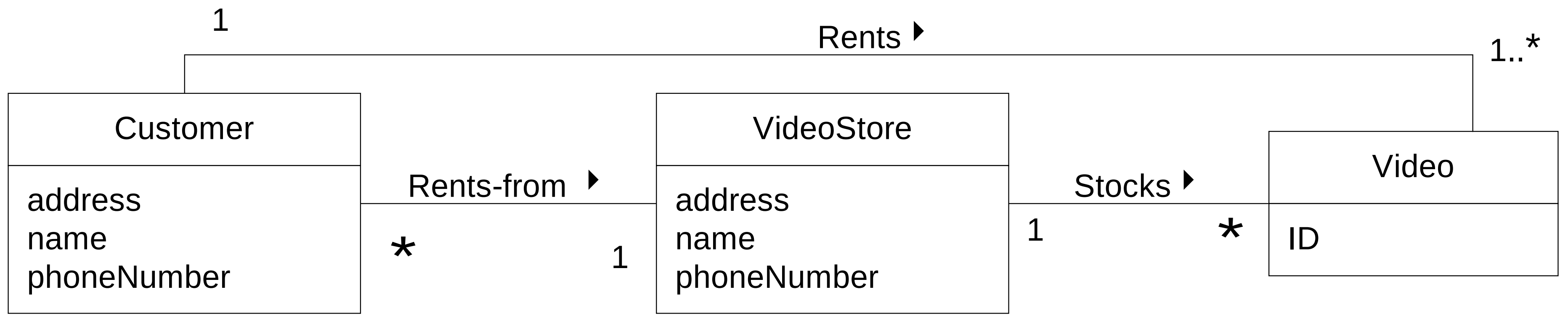
We will use UML to describe the problem domain, describe the activities that occur, and eventually describe the software classes

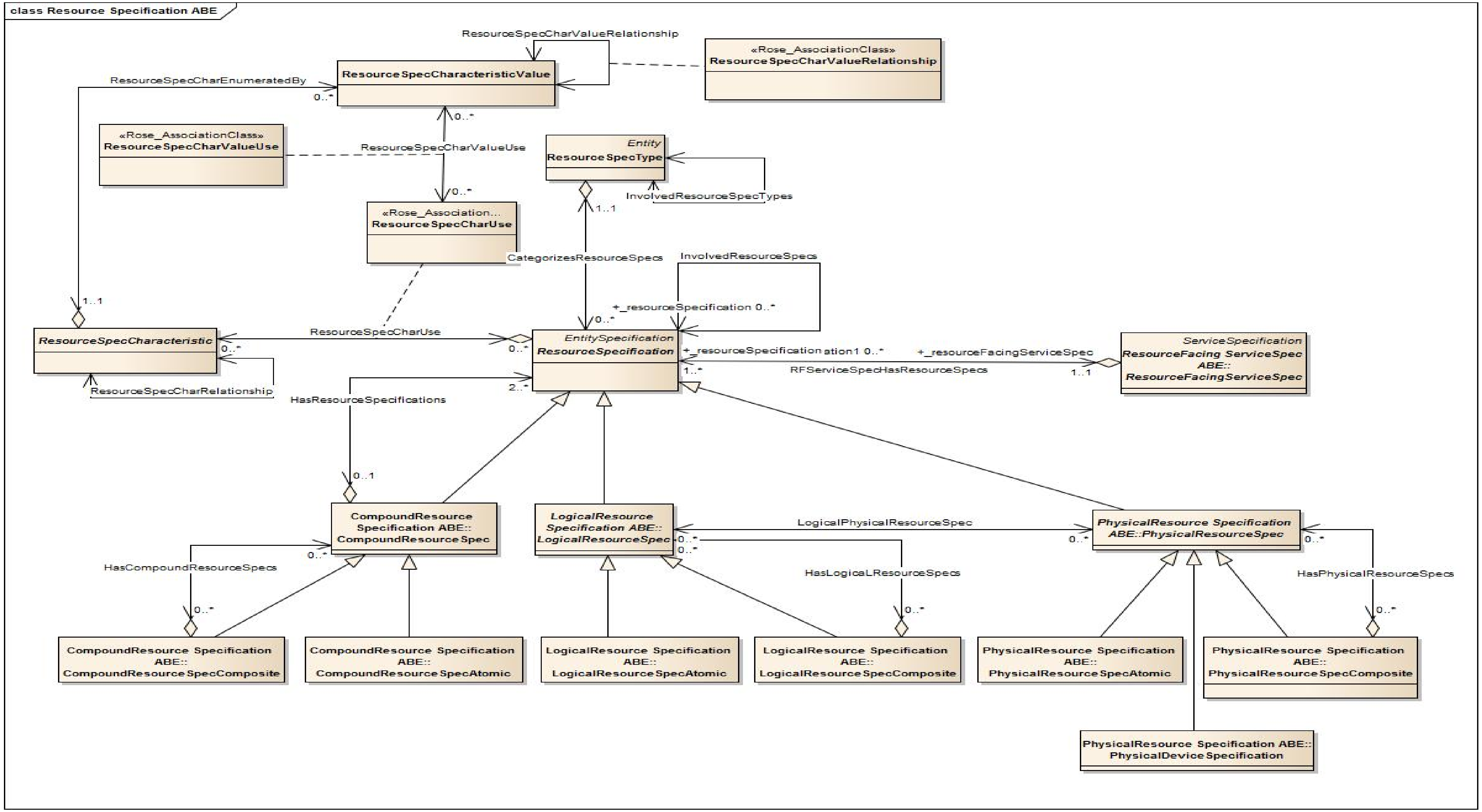
Since it is a language, UML has specific rules, and we will see these later in the course

You need to be able to read UML diagrams, as well as create them

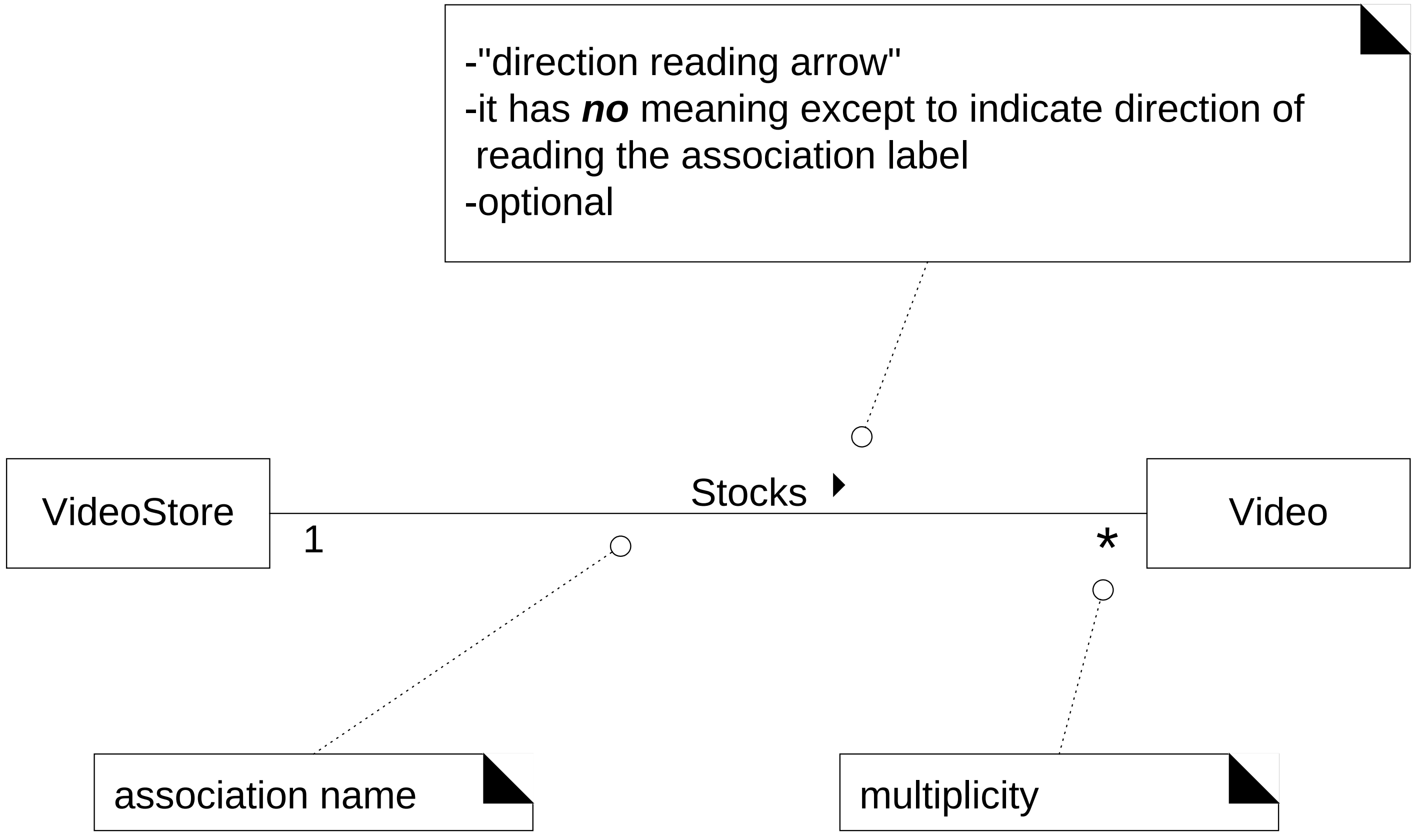
Here are some examples (we will learn more about how to create these diagrams later …)

UML

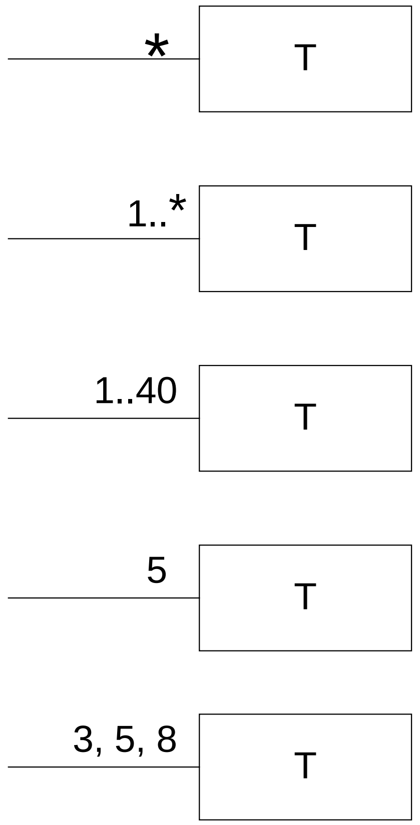

1
Rents
1..*
Customer
address
name
phoneNumber
Rents-from
*
1
VideoStore
address
name
phoneNumber
Stocks
1
*
Video
ID


class Resource Specification ABE
ResourceSpecCharValueRelationship
«Rose_AssociationClass» ResourceSpecCharValueRelationship
ResourceSpecCharEnumeratedBy
ResourceSpecCharacteristicValue
0..*
0..*
«Rose_AssociationClass» ResourceSpecCharValueUse
ResourceSpecCharValueUse
0..*
Entity ResourceSpecType
InvolvedResourceSpecTypes
1..1
«Rose_Association... ResourceSpecCharUse
CategorizesResourceSpecs
InvolvedResourceSpecs
+_resourceSpecification 0..*
1..1
0..*
ResourceSpecCharacteristic
ResourceSpecCharUse
0..*
0..*
EntitySpecification ResourceSpecification
+_resourceSpecification ation1 0..*
+_resourceFacingServiceSpec
1..*
RFServiceSpecHasResourceSpecs
1..1
ServiceSpecification ResourceFacing ServiceSpec ABE:: ResourceFacingServiceSpec
ResourceSpecCharRelationship
2..*
HasResourceSpecifications
0..1
CompoundResource Specification ABE:: CompoundResourceSpec
LogicalResource Specification ABE:: LogicalResourceSpec
LogicalPhysicalResourceSpec
0..*
0..*
0..*
PhysicalResource Specification ABE::PhysicalResourceSpec
0..*
HasCompoundResourceSpecs
0..*
HasLogicaLResourceSpecs
HasPhysicalResourceSpecs
0..*
0..*
0..*
CompoundResource Specification ABE:: CompoundResourceSpecComposite
CompoundResource Specification ABE:: CompoundResourceSpecAtomic
LogicalResource Specification ABE:: LogicalResourceSpecAtomic
LogicalResource Specification ABE:: LogicalResourceSpecComposite
PhysicalResource Specification ABE:: PhysicalResourceSpecAtomic
PhysicalResource Specification ABE:: PhysicalResourceSpecComposite
PhysicalResource Specification ABE:: PhysicalDeviceSpecification

UML


-"direction reading arrow"
-it has ***no*** meaning except to indicate direction of reading the association label
-optional
VideoStore
Stocks
Video
1
*
association name
multiplicity

UML



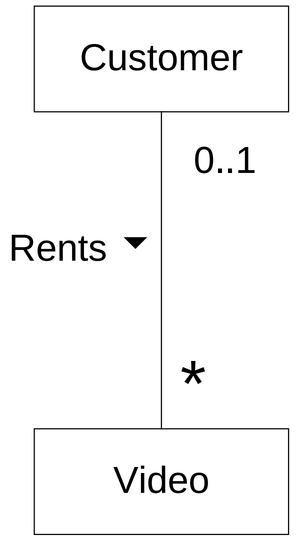
zero or more; "many"

one or more

one to forty

exactly five

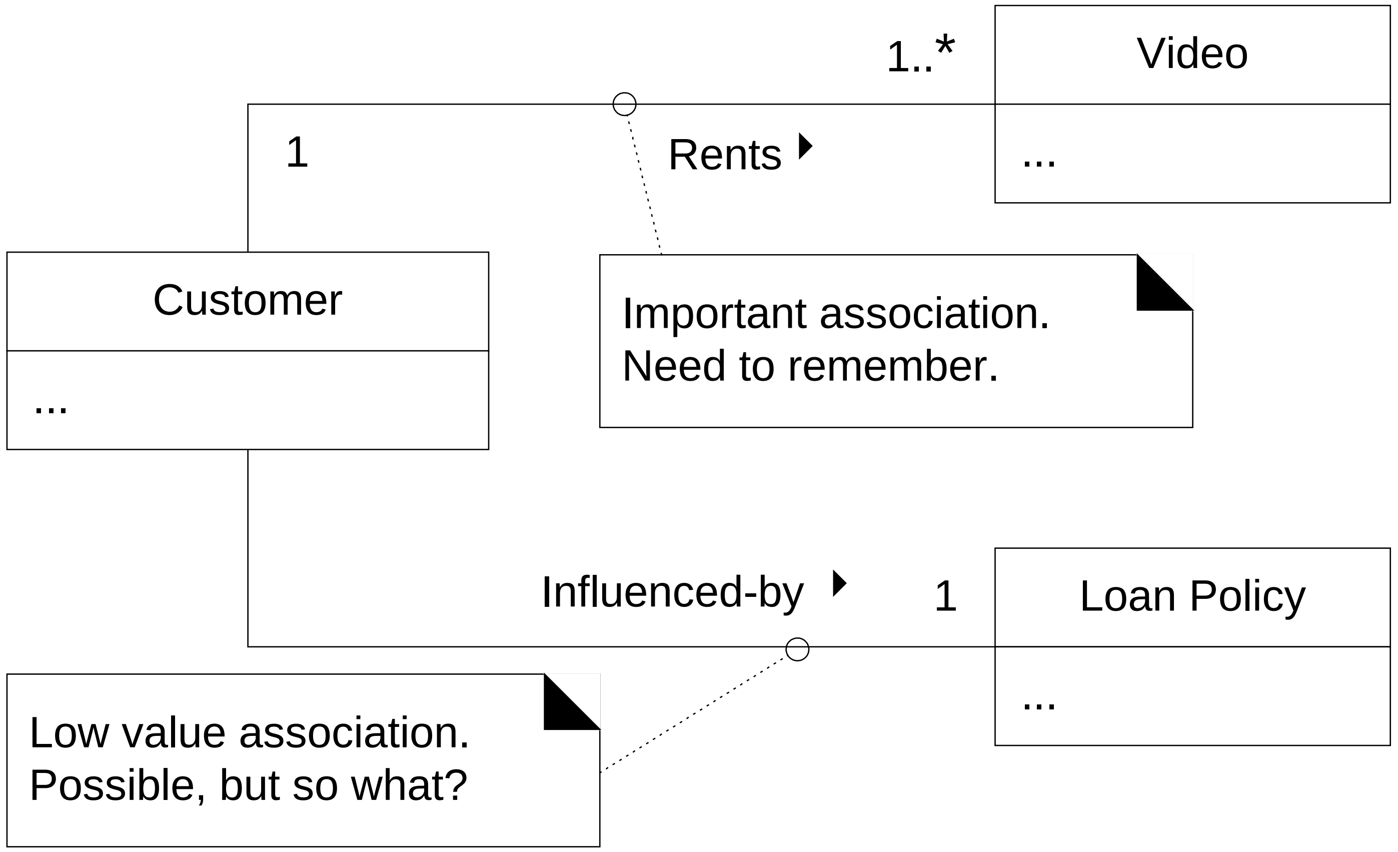
exactly three, five or eight



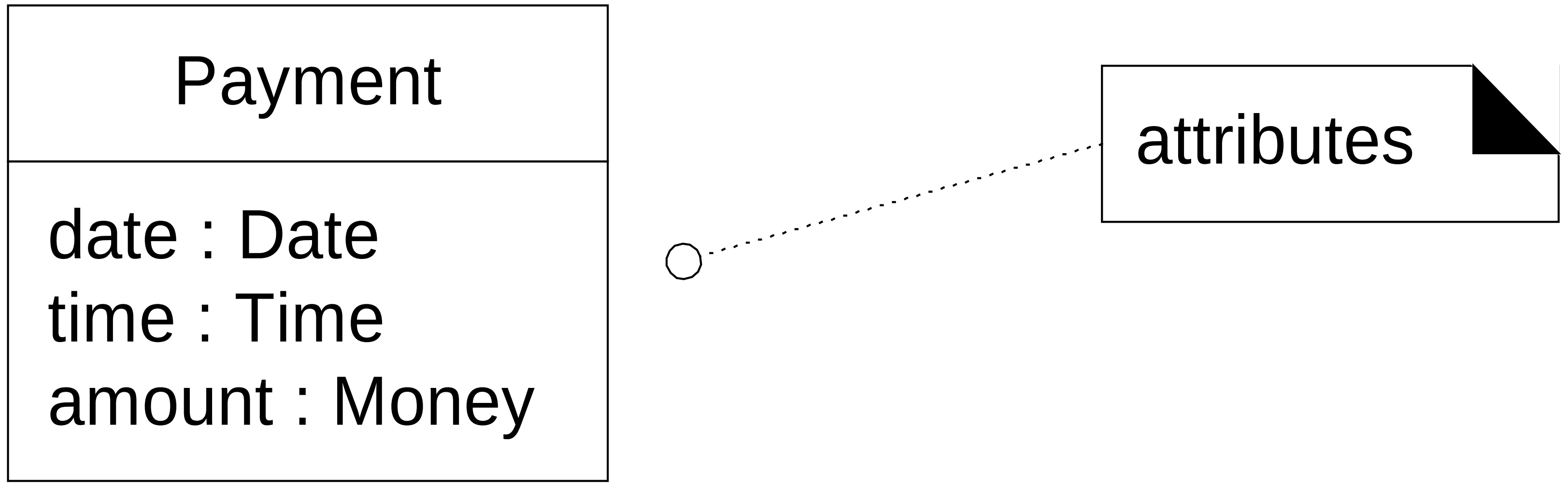
One instance of a Customer may be renting zero or more Videos.

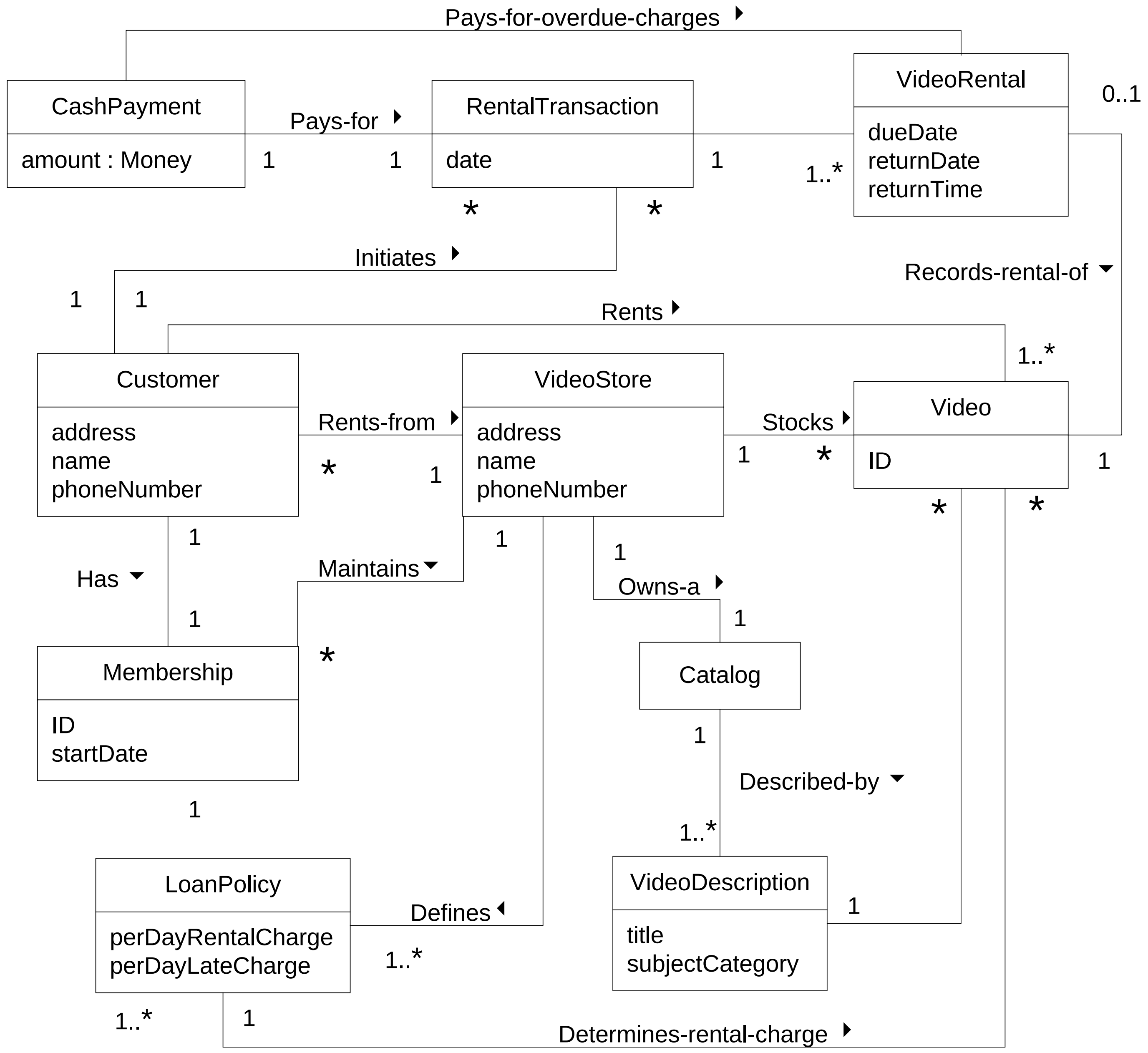
One instance of a Video may be being rented by zero or one Customers.

UML


1..*
Video
...
1
Rents
Customer
...
Important association.
Need to remember.
Influenced-by
1
Loan Policy
...
Low value association.
Possible, but so what?

UML




Pays-for-overdue-charges ▸
CashPayment
amount : Money
Pays-for ▸
1
1
RentalTransaction
date
VideoRental
dueDate
returnDate
returnTime
0..1
1
1..*
*
*
Initiates ▸
Records-rental-of ▾
1
1
Rents ▸
1..*
Customer
address
name
phoneNumber
Rents-from ▸
*
1
VideoStore
address
name
phoneNumber
Stocks ▸
1
*
Video
ID
1
*
*
1
Has ▾
1
Maintains ▾
1
1
Owns-a ▸
1
Membership
ID
startDate
*
Catalog
1
Described-by ▾
1..*
1
LoanPolicy
perDayRentalCharge
perDayLateCharge
Defines ◂
1..*
VideoDescription
title
subjectCategory
1
1..*
1
Determines-rental-charge ▸

Example: The Dice Game

We will give an example of OOA/OOD for the following problem: We want to design an program that simulates a dice game. In the game, the player rolls two dice, and if the sum is seven, the player wins. Otherwise the player loses.

Step One: Define the Use Case.

In this case, the use case is very simple, and can be stated as this:

The Player plays the DiceGame. The DiceGame includes two Dice. The Player requests to roll the Dice. If the total of the two Dice is seven, the Player wins. Otherwise, the Player loses.

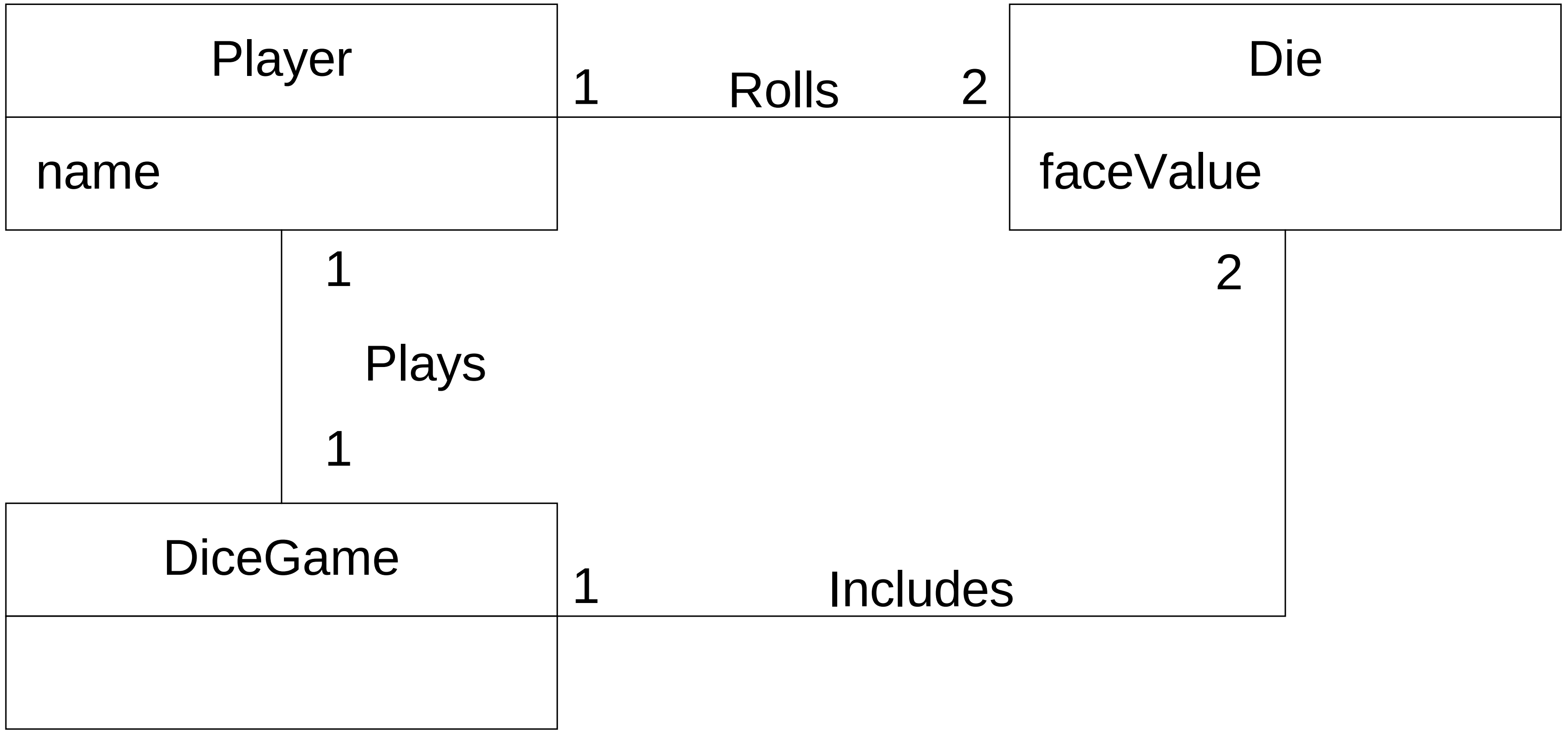
Step two: Define the Domain Model

In this step, we describe the domain of the problem in terms of objects, and try to identify the associations between those objects

We may add the noteworthy attributes of the objects

Note that this is a *conceptual model* – not a design class diagram

The Dice Game: Domain Model



The Dice Game: Object Responsibilities and Interactions

Step 3: We begin to define the software classes, and assign responsibilities

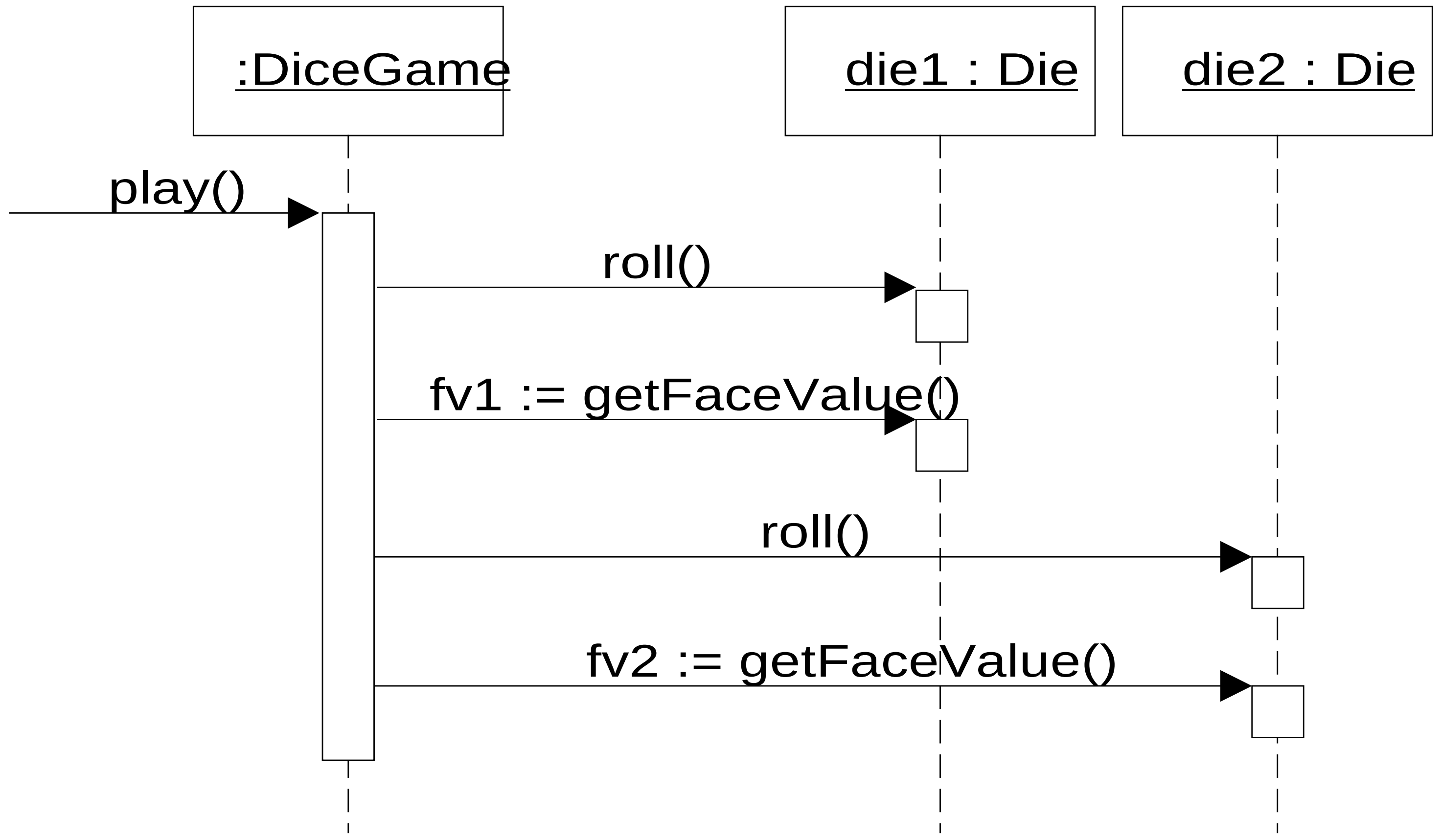
We can define “interaction diagrams” for software classes that will represent the objects we defined in Step 2

Note that at this step, we begin to deal with the real software classes (more specifically, instances of the real classes).

These will be based upon, but may not exactly match up with, the domain concepts or objects we identified in Step 2.

For example, there is no class for Player, since this is not part of the app we are developing

The Dice Game: Interaction Diagram



The Dice Game: Design Class Diagrams

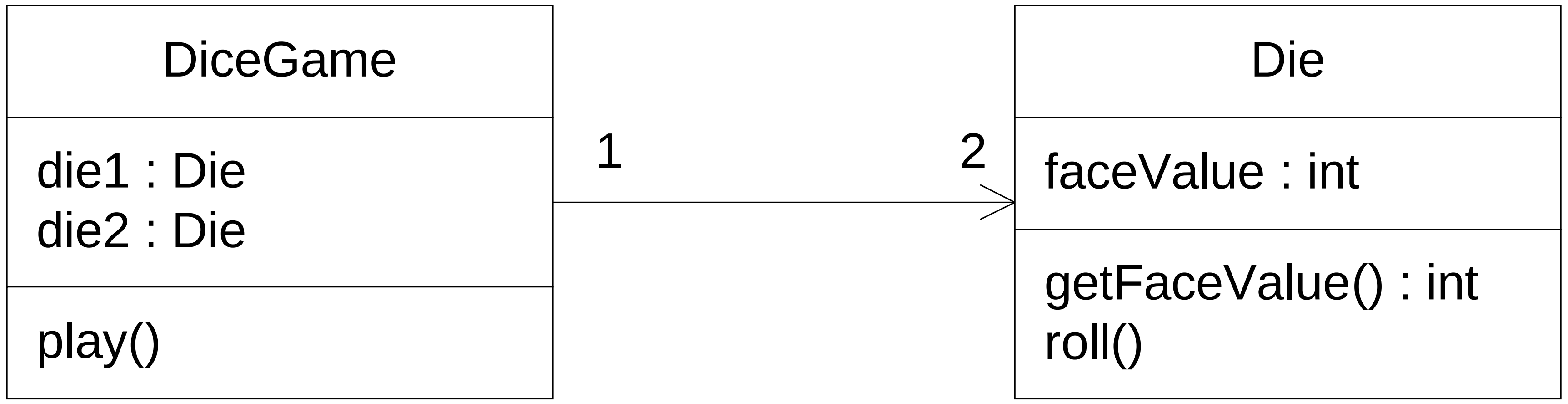
Step 4: We are now ready to design the class diagrams

Using the interaction diagrams, we can define the classes, their attributes, and methods

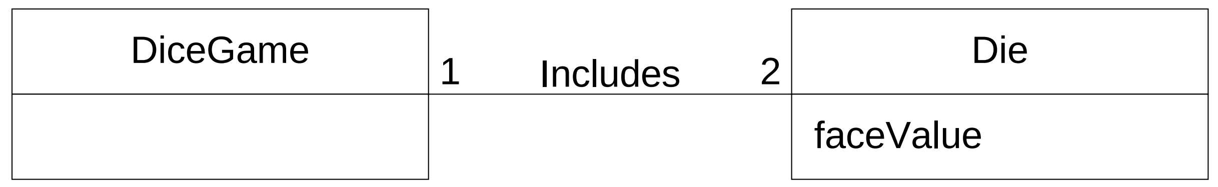
Note: This is not coding. We do not specify how the methods will work, and we may not specify every attribute of the class.

We want to define the important methods and attributes that are needed to make the system work, so that a developer can take this model and build code from it

The Dice Game: Class Design Diagram

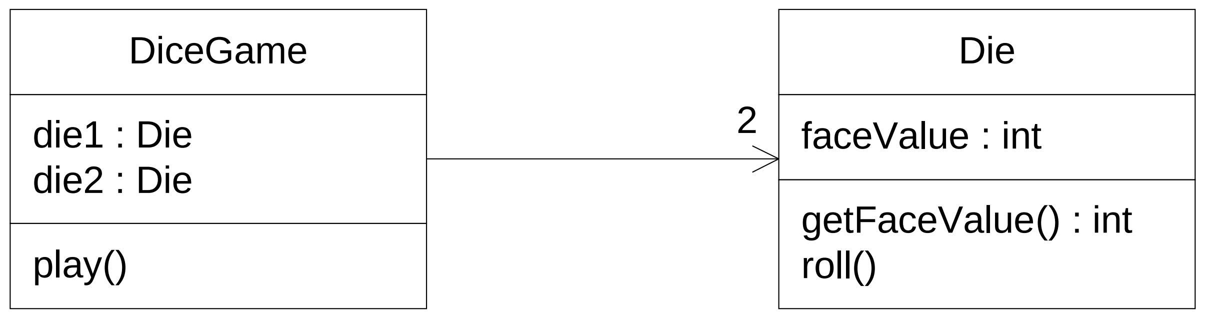


UML: Conceptual versus Specification Perspective



Conceptual Perspective (domain model)

Raw UML class diagram notation used to visualize real-world concepts.



Specification or Implementation Perspective (design class diagram)

Raw UML class diagram notation used to visualize software elements.

iterative, evolutionary, and agile

Iterative Development

Suppose you were assigned to write a movie script for a company. They know what they want, in terms of what kind of movie, how long, setting, etc., but they need you to fill in the details. How would you do it?

You could spend a lot of time talking to them, getting as much information as possible, and then write the script and send it to them and hope you nailed it …

- Risky: Chances of getting it all right are slim, and if you missed you need to go back and start making changes and edits, which can be complicated if you want the story line to work

You could also start with a draft, and send the incomplete version to them for feedback

- They would understand that this is not finished, but just a draft.
- They provide feedback, you add more details, and the cycle continues
- Eventually, you end up with the finished product that is close to what they wanted

This is how iterative development works

Unified Process (UP)

A widely adopted process for building, deploying, and possibly maintaining software

- Flexible and open, can adopt practices from other methods such as Scrum

We will concentrate on Agile UP

Iterative and evolutionary development

- Development is organized into a series of short, fixed length mini-projects called iterations
- Note – emphasizes early programming and testing of partial systems, even before all requirements are finalized!
- Direct contrast to sequential methods (i.e. Waterfall), where coding is not started until the requirements phase is finished.

Each iteration enlarges and refines the project, with continual feedback and adaption as the main drivers

Iterative Methodology

Doesn’t this contradict what we learned in Chapter 1 … analysis first, then design?

- We will see more on this throughout the course, but for now understand that iterative development does not simply mean “start coding”. There is analysis and design work first.

Each iteration is fixed length, called *timeboxed*

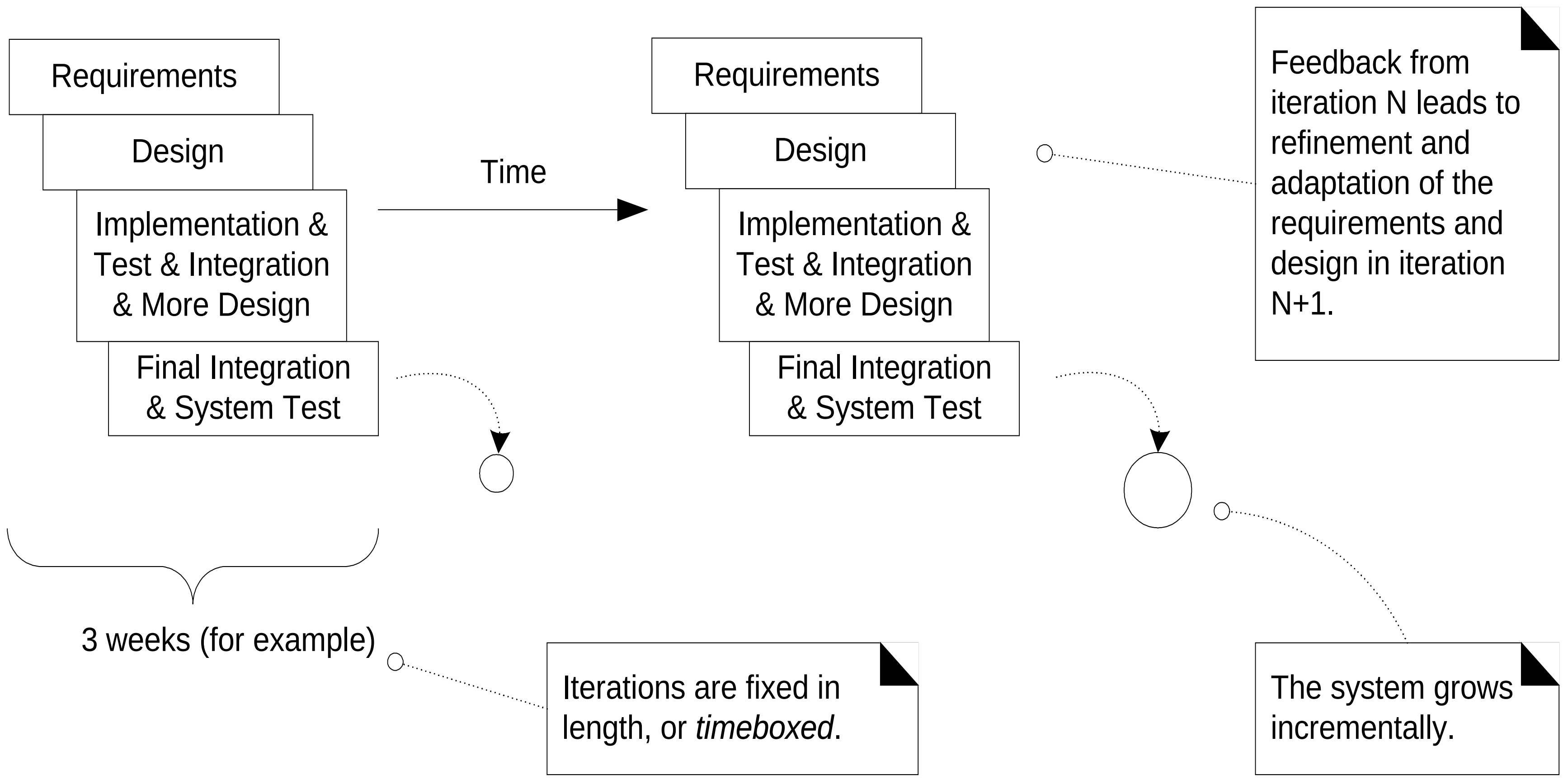
Typical iteration: Spend the first day analyzing the current state of the project, defining goals for the current iteration, and then working on design in small teams

Then teams work on implementing new code, testing, more design discussions, and conducting daily builds of partial system

- Usually involves demonstrations and evaluations with stakeholders (customers or customer reps)

Also planning for next iteration – each iteration may involve analysis and design work

The Iterative Process


Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
Time
Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
3 weeks (for example)

Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1.

Iterations are fixed in length, or *timeboxed*.

The system grows incrementally.

Embrace Change

Rather than try to “freeze” the requirements so code can be developed to rigid design, accept the fact that change will occur and use it in the development process

- Maybe the customer changes mind after seeing early partial system builds

This has become very popular because it reflects the way the real world works

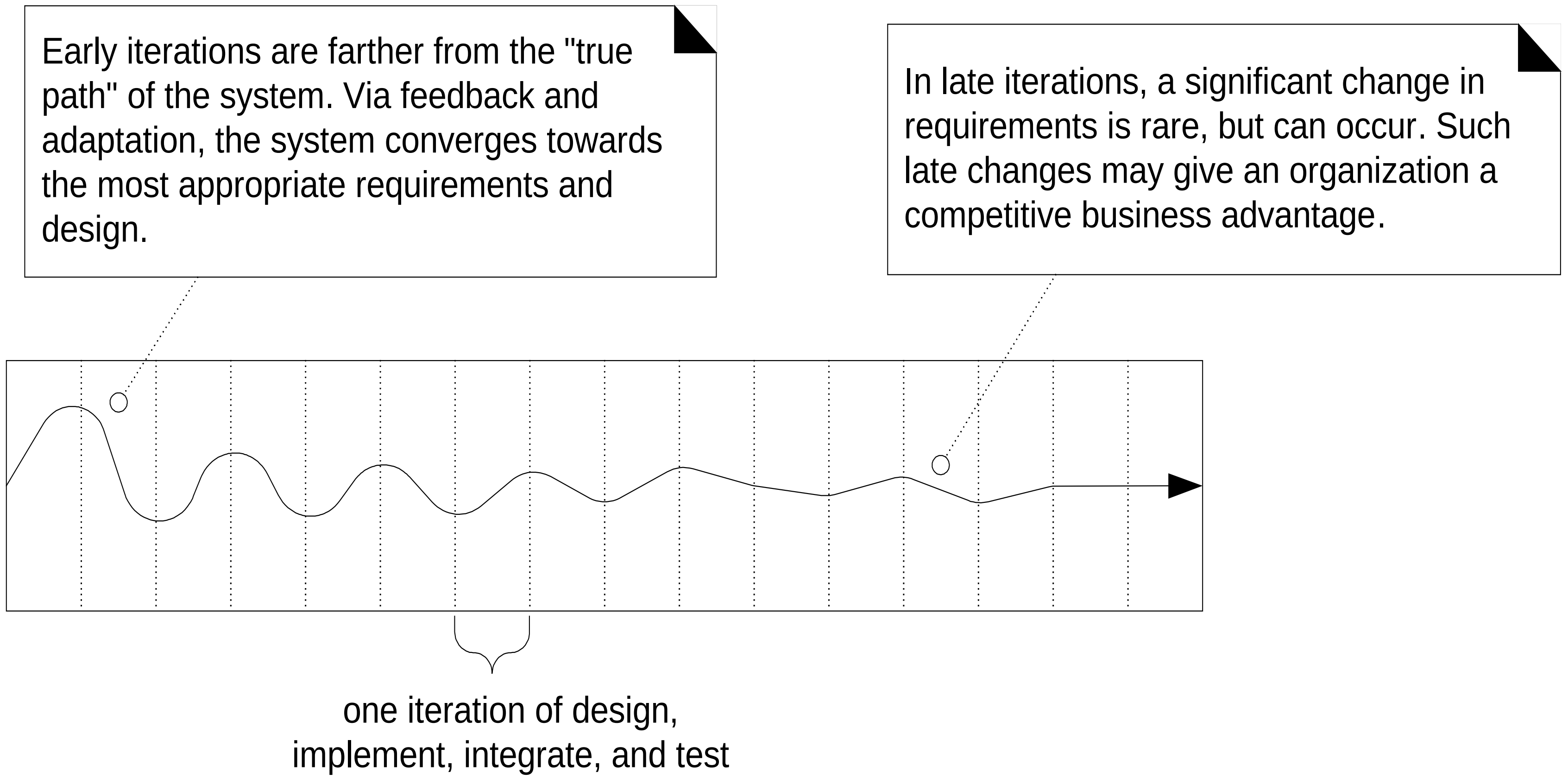
- Also, coding technology has enhanced code generation methods

OOA/OOD is critical to this approach, since having well defined objects makes it easier to add to or modify the system

Note that this does not mean there are no requirements, or that changes should be *encouraged*

- Beware of “feature creep” – we will see more later
- This is structured change, not chaos

Iteration Convergence



Early iterations are farther from the "true path" of the system. Via feedback and adaptation, the system converges towards the most appropriate requirements and design.
In late iterations, a significant change in requirements is rare, but can occur. Such late changes may give an organization a competitive business advantage.
one iteration of design, implement, integrate, and test

The Benefits

- Early feedback – invaluable!
- Better success rate for projects
  - More likely that the customer will get what they want in the end
- If process is properly executed, early mitigation (resolution) of high risks, rather than later
- By breaking the project into clearly defined iterations (cycles), the complexity is much more manageable
  - Don’t need to solve the entire project first – no paralysis by analysis
  - Again, natural fit with OOA/OOD
- Constant feedback and input from the customer improves the end product

Waterfall Lifecycle

- Probably an attempt to apply standard product development methods to software
- Fully define requirements (and usually design) before coding
- Why is Waterfall so prone to fail?
  - Software development is not mass production – highly unlikely that all requirements are fully understood up front
  - Requirement are also not as stable as we would like, especially for large complex projects
  - Change Request process works, but can be cumbersome and slow
  - Note it is possible to change requirements later in the project for Waterfall, but it is hard and slow
- Try to avoid combining this approach with an iterative approach
  - Don’t try to identify all use cases or do complete OOA before coding can start
  - Software design and implementation becomes more complex as understanding of the system increases

Example

Before first iteration, work with customer in a Requirements Workshop to identify a few critical Uses Cases

- Highest risk, most important features, use cases
- Do a deep functional analysis of these use cases

Plan the iteration, i.e. identify which of the selected use cases will be addressed

Iteration 1:

- First couple of days are for OOA/OOD of the assigned tasks
- Remainder of the iteration (x number of weeks): Write code, test, integrate
- If it appears that the original goals for the iteration cannot be met, “de-scope”
- Freeze code for the iteration, demo to customer, get feedback
- Near end of iteration, return to use cases, begin to add to requirements/tasks based upon feedback
- Plan Iteration 2, and continue

Example

For the first 4-5 iterations, there will be much requirements analysis and refinement

This is the *elaboration phase* – after it is completed, we have maybe 90% of the requirements identified, but only about 10% of the code written.

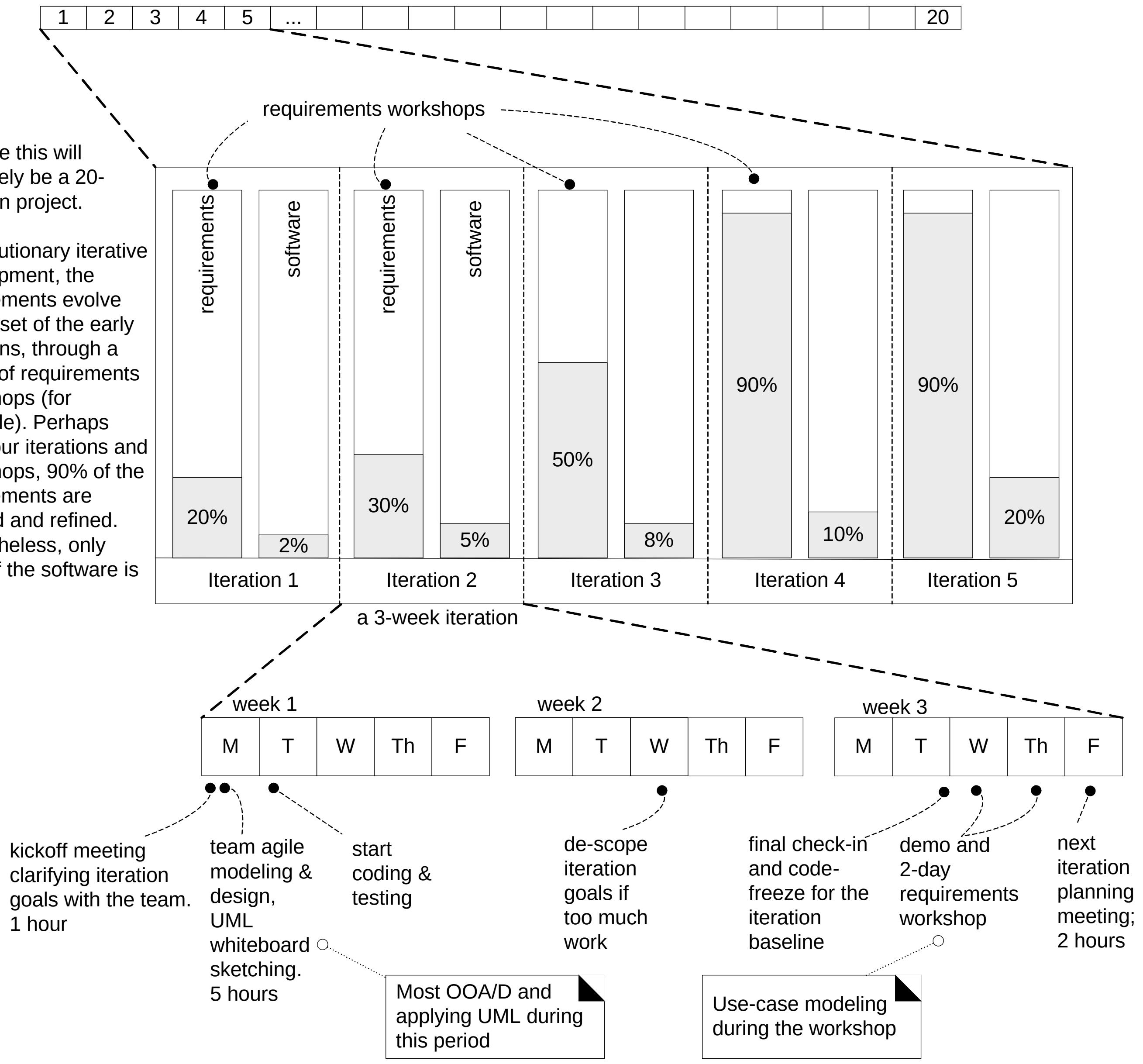
This is the time to estimate the time and effort the remainder of the project will take

- Should be based upon the actual work done to this point, as so should be accurate

After this, fewer requirements workshops are needed, although the requirements are never considered “frozen”

This approach is considered *risk-driven* and *client-driven*

- Goal of the earliest iterations is to identify and drive down highest risks, build features customer cares most about


1
2
3
4
5
...
20
requirements workshops
requirements
software
20%
2%
30%
5%
50%
8%
90%
10%
90%
20%
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
a 3-week iteration
week 1
week 2
week 3
M
T
W
Th
F
kickoff meeting clarifying iteration goals with the team. 1 hour
team agile modeling & design, UML whiteboard sketching. 5 hours
start coding & testing
de-scope iteration goals if too much work
final check-in and code-freeze for the iteration baseline
demo and 2-day requirements workshop
next iteration planning meeting; 2 hours
Most OOA/D and applying UML during this period
Use-case modeling during the workshop

Imagine this will ultimately be a 20-iteration project.

In evolutionary iterative development, the requirements evolve over a set of the early iterations, through a series of requirements workshops (for example). Perhaps after four iterations and workshops, 90% of the requirements are defined and refined. Nevertheless, only 10% of the software is built.

Agile Methods and Attitudes

Stress *agility* – rapid and flexible response to change

The Agile Manifesto:

- Individuals and interactions *over* processes and tools
- Working software *over* comprehensive documentation
- Customer collaboration *over* contract negotiation
- Responding to change *over* following a plan

Any iterative process can be applied in the Agile spirit

The Agile Principles

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

The Agile Principles

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile Modeling

- We create models (or UML drawings) to help understand the project
  - Not just documentation, or a blueprint to be handed to the software developer
- Modeling is very important for Agile methods, and is a key tool for <u>communication</u>
- Don’t “over-model”
  - Not every class needs a UML drawing
  - Some simpler or more obvious parts of the problem can be deferred until coding starts
  - Apply modeling and UML drawing for the trickier or more complicated parts
- Models and diagrams do not need to be complete, or include every detail – they will be inaccurate, and that is OK
- Usually done at a whiteboard, with 2-4 people

UP – The Key Practices

Tackle high-risk and high-value issues in early iterations

Continuously engage users for evaluation and feedback, requirements evolution

Build a cohesive, core architecture early on

Continuously verify quality: test early and often, and realistically!

Apply use cases where appropriate

Utilize visual modeling (UML)

Carefully manage requirements

Practice change request and configuration management

UP Phases

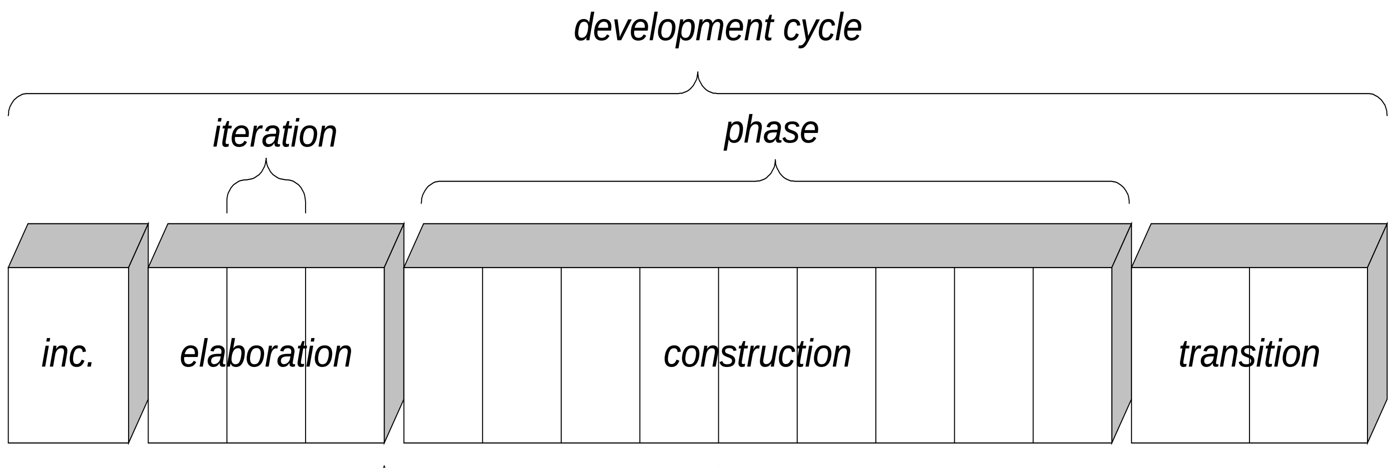
Inception: approximate the vision, business case, scope, *vague* estimates

Elaboration: Refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, realistic estimates

Construction: Iterative implementation of the remaining low risk and easier elements, preparation for deployment

Transition: Beta testing, deployment

UP Phases


development cycle
iteration
phase
inc.
elaboration
construction
transition

The UP Disciplines

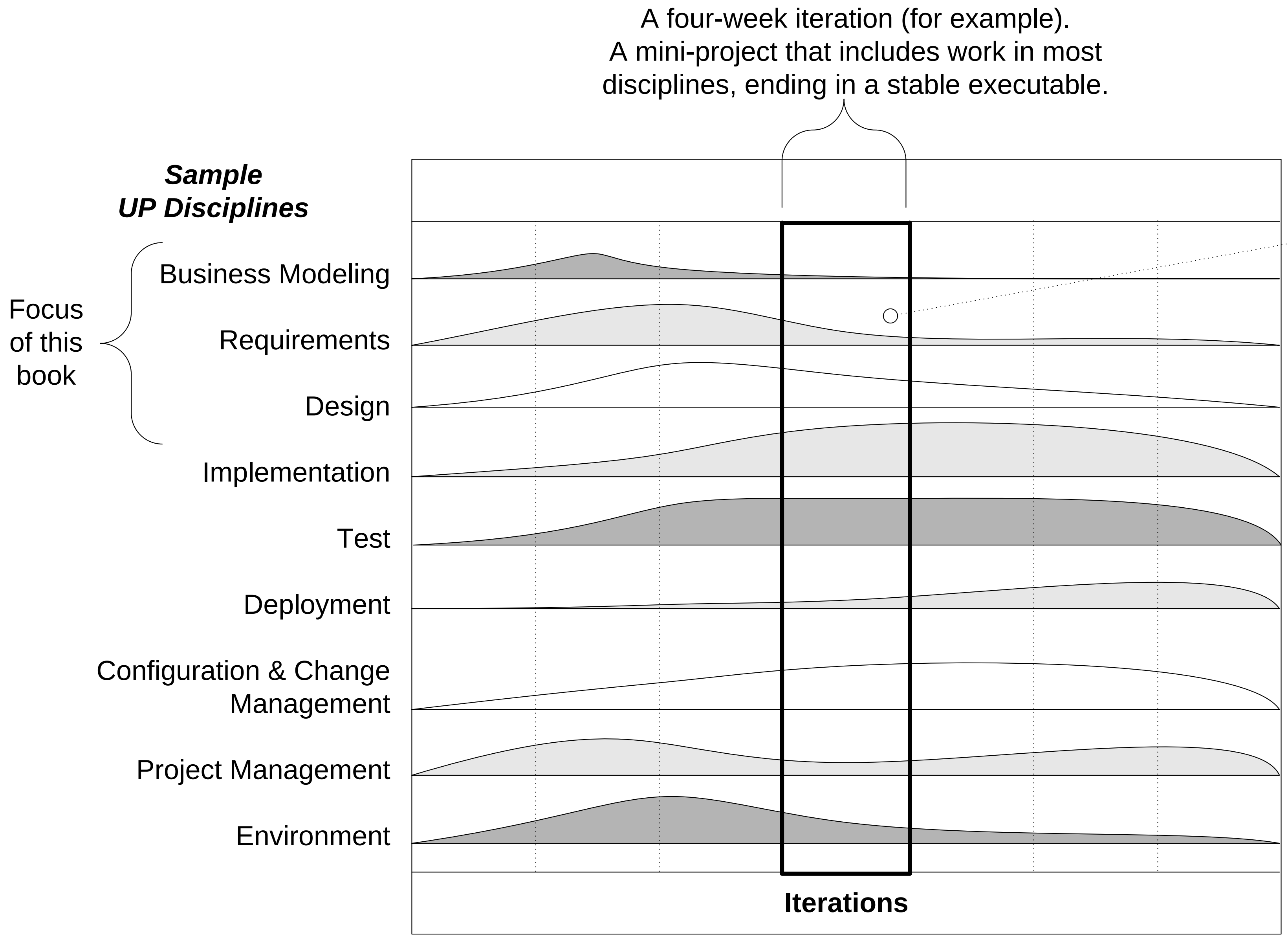
Disciplines are sets of activities and related artifacts in one subject area

- An *artifact* is any work product, e.g. code, UML model diagram, etc.

We will concentrate on the following disciplines:

- **Business Modeling**: Development of the Domain Model artifact
- **Requirements**: The Use Case Model and Supplementary Specifications artifacts
- **Design**: The Design Model artifact

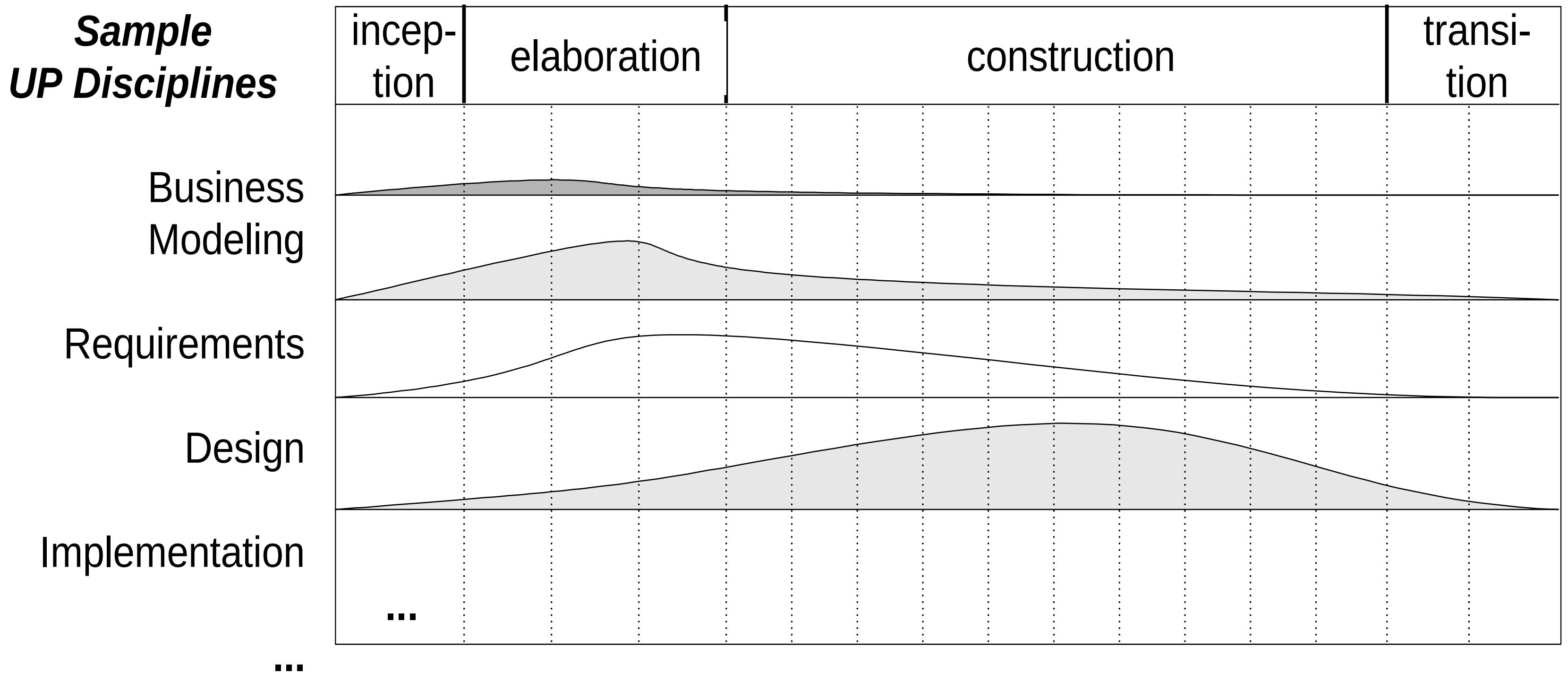
Also, UP recognizes Implementation, Test, Deployment, Configuration and Change Management, Project Management, and Environment (setting up the tools and process environment for the project)



Note that although an iteration includes work in most disciplines, the relative effort and emphasis change over time.

This example is suggestive, not literal.

UP Disciplines and Phases



The relative effort in disciplines shifts across the phases.

This example is suggestive, not literal.

UP Development Case

Most UP activities and artifacts are considered optional – the methodology is designed to be flexible, and hence applicable to many different types of projects

- But some practices and principles – iterative development, continuous testing – are not optional

The choice of which artifacts to use and practices to follow for the project may be captured in a *Development Case* – this is an artifact in the Environment discipline

- This document would capture the expected artifacts to be created, for example

More on Agile Methods

There are many, including Extreme Programming (XP), Dynamic Systems Development Method (DSDM), Feature Driven Development (FDD), Scrum

There has been some criticism:

- Not clear how well this applies to large scale development projects (20+ developers)
- May be hard to outsource, or for teams that are not co-located
- Latest management fad?
- Sometimes massive documentation is required by the customer
- Cost estimation – not sure how much the project will cost until well into it?

The slides are structured in an iterative manner – we will skip around the chapters

When Do Agile Methods Work Best?

Particularly useful is the customer is not clear on what they want …

The iterative approach helps them define the vision/solution as it is built

Only need a good vision to start with

Customer is open to regular communication and feedback – they are willing to be active partners in the design process

Agile may not be the best way to go if the customer knows exactly what they want

- Example: Revising existing application with a clear set of enhancements
- Can still apply the Agile methodology, but the advantage is smaller

The Case Studies We Will Work With

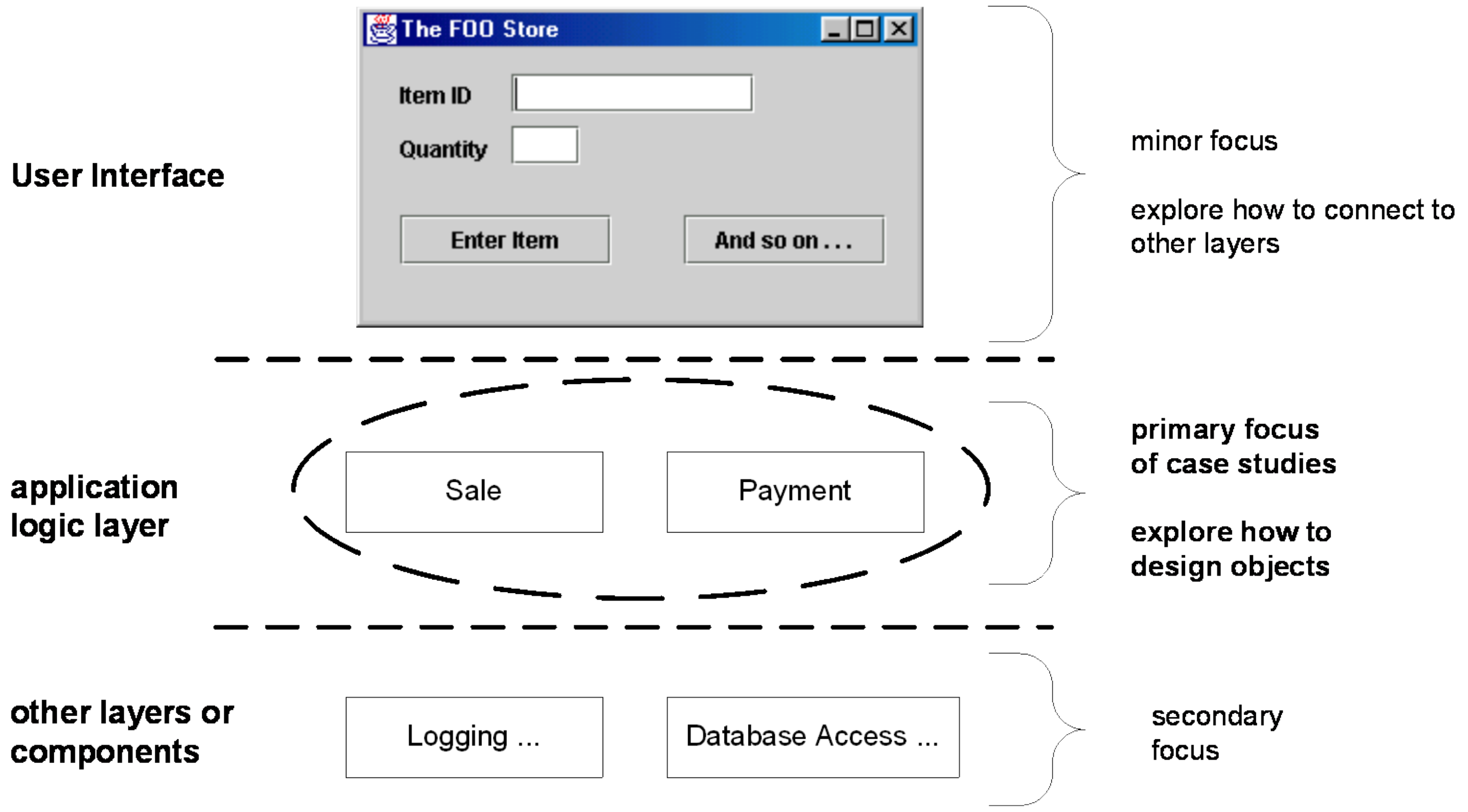
These are example systems that the book and lectures will refer to

NextGen Point of Sale (POS) System: This is a system that runs in a store and allows cashiers to check out items for customers. The customers bring a basket of items to the register, the cashier scans or inputs the item information, and a sale total is calculated and processed

Monopoly Game Simulation: This is an application that simulates the playing of a game of Monopoly. The simulation will track the players moves over a set amount of time.

The Case Studies

We will mostly concentrate on the application logic layer



Case One: The NextGen POS System

Point-of-Sale (POS) system is application used to record sales and secure payment

- Checkout line at store

System includes hardware and software – we will concentrate on the software

Has interfaces to various service apps, like tax calculator and inventory control, and the system should work even if access to these external services is down (i.e., at least allow checkout with cash if the credit card processing interface goes down)

Needs to support multiple client-side interface types, like thin web-browser, touchscreen, wireless phone, etc.

We plan to sell this to many types of businesses which may have different business processing rules – we need flexibility and the ability to customize

Case Two: The Monopoly Game

Software version of Monopoly game

This will run as a simulation; the user will configure the game (i.e. indicate the number of players, etc.), start the simulation, and let the game run to its conclusion.

A trace log will be created to record each move a player makes

The simulation should include the rules of the game, and keep track of the amounts each player earns/loses throughout the game

The simulation should allow the user to select various strategies to be employed by the players in the game

inception, requirements, use cases

What will we learn?

- Inception – what is it?
- How to analyze requirements in iterative development
  - The FURPS+ model, and the UP Requirements artifacts
- How to identify and write Use Cases
- How to apply tests to identify suitable Use Cases
- How to develop Use Cases in iterative development

Inception

Inception is the initial **short** step that is used to establish a common vision and basic scope for the project

Main questions that are often asked:

- What is the overall vision and business case for the project?
- Is it feasible?
- Buy or build?
- Rough cost estimate (order of magnitude)
- Go, no go

We **do not** define all of the requirements in Inception!

- Perhaps a couple of example requirements, use cases

We are **not** creating a project plan at this point

Inception

Goal: Envision the project scope, vision, and business case.

Is there a basic agreement among the stakeholders on the vision, and is it worth investing in a serious *investigation*?

- Note *investigation* versus *development*

Inception is brief

- Decisions on feasibility and go no go *may* have already been made

There may be some simple UML diagrams, and even some basic coding for proof-of-concept prototypes to answer key questions

Evolutionary Requirements

This is where Waterfall and UP part ways …

Requirements are capabilities and conditions to which the system – and more broadly, the project – must conform.

Since UP does not require all requirements to be defined up front, it requires careful *management* of requirements

- “a systematic approach to finding, documenting, organizing, and tracking the changing requirements of the system”

Key difference between Waterfall and UP: UP *embraces* requirements changes

How to find the requirements?

- Different methodologies do this in different ways; Requirements Workshops, Use Cases, etc.

FURPS+

Functional – features, capabilities, security

Usability – human factors, documentation

Reliability – frequency of failure, recoverability, predictability

Performance – response times, throughput, accuracy, availability, resource usage

Supportability – adaptability, maintainability, internationalization, configurability

Plus …

- Implementation, Interfaces, Operations, Packaging, Legal, etc.

Often used: functional (behavioral) versus non-functional (everything else)

Quality Requirements: usability, reliability, performance, and supportability

Requirements Organization: UP Artifacts

Use-Case Model: The use cases will primarily capture the functional requirements, i.e. how the system behaves

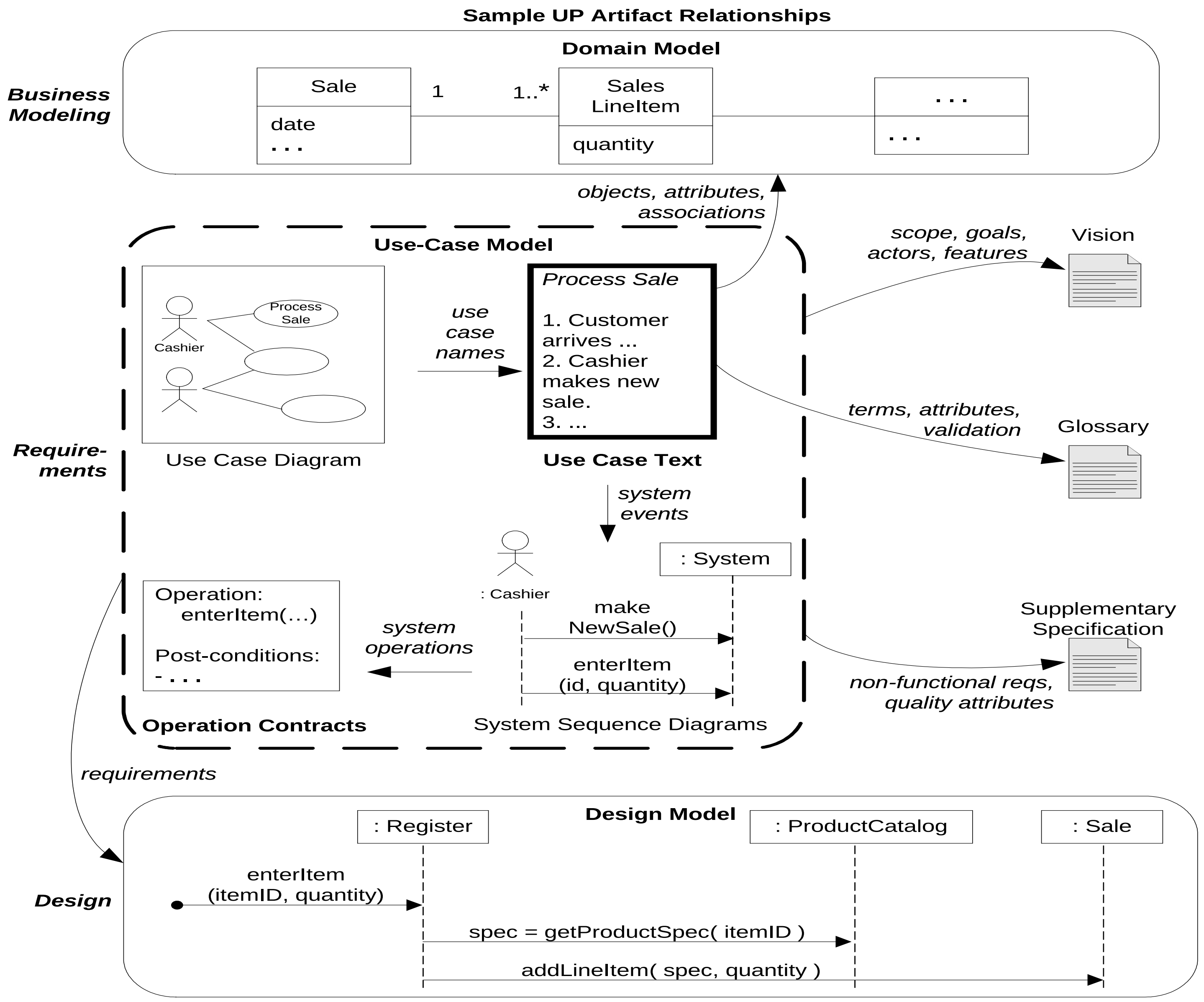
Supplementary Specification: Non-functional requirements (e.g. performance) and any functional features not captured by the Use-Case Model

Glossary: Noteworthy terms, but can include data dictionary (which may include any requirements on data rules, e.g.)

Vision: May capture high-level requirements

Business Rules (Domain Rules): These usually transcend any one project, and so may be captured in one place for use by several projects. Think regulatory requirements.

This diagram shows the relationships between various artifacts in UP and the main disciplines we will consider in this course – Business Modeling, Requirements, and Design



The Use-Case Model Artifact

… is part of the Requirements discipline in UP

… is composed of the use case text documents

… may include UML diagrams of the use cases, which provide context diagrams of the system

… is essential to OOA/OOD, although OOA/OOD is not used to develop the Use-Case Model

Use cases are requirements – primarily functional (behavioral)

Use cases provide an essential understanding of how the end user will use the system, and as such are critical for proper requirements definition

Key Features

Use cases are **text**, not diagrams

- We may construct diagrams from the text cases later

These are created as stories, and functional requirements are derived from them

Actors: something that exhibits behavior in the system (not just a person)

Scenario: A specific sequence of actions and interactions between the actors and the system (use case instance)

- One particular story of using the system

Use Case (informal): A collection of related success and failure scenarios that describe an actor using a system to achieve a goal.

Use Case (formal, RUP): A set of use-case instances (scenarios), where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.

Key Features

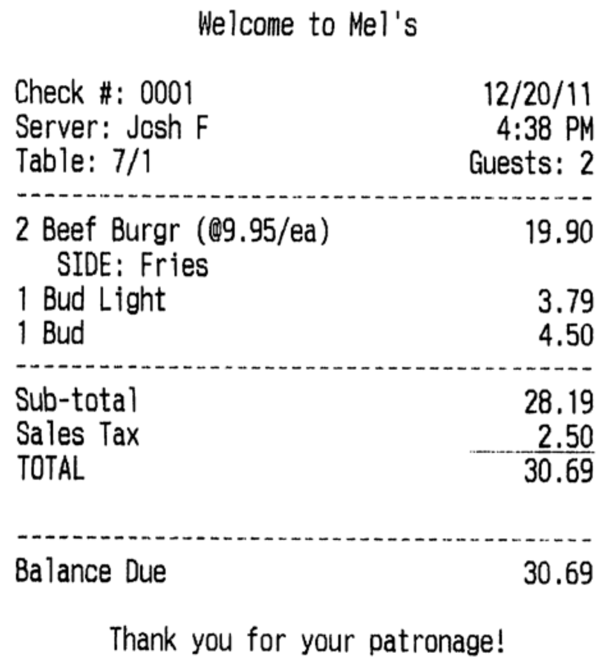
One of the key skills you will need in use case development and analysis: Identify the nouns (things, objects) in the use case, and the verbs (actions).

The nouns will eventually give you clues about the objects in the system

The verbs will give you clues about their behavior, their responsibilities

For example, suppose we have a use case with the sentence "The line item is added to the sale."

- We note the nouns *line item* and *sale* – these are likely objects in the system
- We note that something must have the responsibility of *adding* the line item to the sale … this is something our system must do, it is a *functional requirement*.



```
              Welcome to Mel's

Check #: 0001                     12/20/11
Server: Josh F                     4:38 PM
Table: 7/1                       Guests: 2
------------------------------------------
2 Beef Burgr (@9.95/ea)              19.90
   SIDE: Fries
1 Bud Light                           3.79
1 Bud                                 4.50
------------------------------------------
Sub-total                            28.19
Sales Tax                             2.50
TOTAL                                30.69

------------------------------------------
Balance Due                          30.69

         Thank you for your patronage!
```

Example: ATM Session Use Case

A session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her PIN, and is then allowed to perform one or more transactions, choosing from a menu of possible types of transaction in each case. After each transaction, the customer is asked whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session ends. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being retained in the machine.

The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type.

Example: ATM Session Use Case

A **session** is started when a **customer** inserts an **ATM card** into the **card reader slot** of the **machine**. The **ATM** pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged **stripe**, the card is ejected, an **error screen** is displayed, and the session is aborted.) The customer is asked to enter his/her **PIN**, and is then allowed to perform one or more **transactions**, choosing from a **menu** of possible types of transaction in each case. After each transaction, the customer is asked whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session ends. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being retained in the machine.

The customer may abort the session by pressing the **Cancel key** when entering a PIN or choosing a transaction type.

Example: ATM Session Use Case

A session is **started** when a customer **inserts** an ATM card into the card reader slot of the machine. The ATM **pulls** the card into the machine and **reads** it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is **ejected**, an error screen is **displayed**, and the session is **aborted**.) The customer is **asked** to enter his/her PIN, and is then **allowed** to **perform** one or more transactions, **choosing** from a menu of possible types of transaction in each case. After each transaction, the customer is **asked** whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session **ends**. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being **retained** in the machine.

The customer may abort the session by **pressing** the Cancel key when **entering** a PIN or choosing a transaction type.

Actors and Use Case Formats

Actors: Anything with behavior, including the system itself

- Primary (has goals fulfilled by the system), Supporting (provides a service), Offstage (interest in the behavior of the system, but not primary of supporting – like government agency, or a monitoring system)
- These do not define the objects

Use Case types:

- Brief or casual (short text descriptions, either one or a few paragraphs)
- Fully Dressed: Detailed, usually follow a template

Use Case Template

Use Case Section	Comment
Use Case Name	Starts with verb, unique, sometimes number
Scope	Identifies the system under design
Level	User-goal level, but may be subfunction if these substeps are used by many use cases
Primary Actor	Actor that calls upon the system to fulfill a goal
Stakeholders and Interests List	Very important – lists all stakeholders in the scenario, and what they expect the system to do. Will identify the behaviors.
Preconditions, Success Guarantees	Conditions that are relevant and considered true at the start of the use case; what must be true upon completion of the scenario

Use Case Template

Use Case Section	Comment
Main Success Scenario	A record of the steps of a successful scenario, including interaction between actors, validation (by system), and state change by system. Steps are usually numbered.
Extensions	All other scenarios that may be branched to off the main success scenario; numbered according to main success scenario steps, and often this section is larger. May refer to another use case
Special Requirements	Non-functional requirements, quality attributes, constraints
Technology and Data Variations List	Any obvious technology or I/O constraints, data constraints
Misc	Anything else

Guidelines (Fully Dressed)

Defer all conditional branches to the Extensions section

Write in “essential style” – leave out user interface specifics, focus on actor intent

Write terse statements

Use “black box” thinking: “System records the sale”, not “System accesses database and generates SQL INSERT …”

- Do not design!

Role play – become the actor, take the actor perspective

Example: ATM Withdrawal (Fully Dressed)

Use Case Name: Withdraw Money From ATM

Scope: ATM System

Level: User-goal

Actors: Customer (Primary), Bank (Supporting), ATM (Supporting)

Stakeholders:

- Customer: Get cash
- Bank: Provide cash, properly record transaction

Preconditions: There is an active network connection to the Bank, the ATM has cash

Example: ATM Withdrawal

Main Success Flow:

1. The use case begins when Bank Customer inserts their Bank Card.
2. Use Case: Validate User is performed.
3. The ATM displays the different alternatives that are available on this unit. [See Supporting Requirement SR-xxx for list of alternatives]. In this case the Bank Customer always selects "Withdraw Cash".
4. The ATM prompts for an account. See Supporting Requirement SR-yyy for account types that shall be supported.
5. The Bank Customer selects an account.
6. The ATM prompts for an amount.
7. The Bank Customer enters an amount.

Example: ATM Withdrawal

Main Success Flow:

8. Card ID, PIN, amount and account is sent to Bank as a transaction. The Bank Consortium replies with a go/no go reply telling if the transaction is ok.

9. Then money is dispensed.

10. The Bank Card is returned.

11. The receipt is printed.

12. The use case ends successfully.

Example: ATM Withdrawal

Alternate flows (details left out):

Invalid User

Wrong account

Wrong amount

Amount Exceeds Withdrawal Limit

Amount Exceeds Daily Withdrawal Limit

Insufficient Cash

No Response from Bank

Money Not Removed

Quit

Example: ATM Withdrawal

Post-Conditions (Success Guarantee):

Successful Completion

The user has received their cash and the internal logs have been updated.

Failure Condition

The logs have been updated accordingly.

Special Requirements:

[SpReq:WC-1] The ATM shall dispense cash in multiples of $20.

[SpReq2:WC-2] The maximum individual withdrawal is $500.

[SpReq:WC-1] The ATM shall keep a log, including date and time, of all complete and incomplete transactions with the Bank.

Example: Data Entry System

We are designing a data entry system. The primary actor is the data entry worker, who will enter information into a user interface to create and update records for the client. During a Requirements Workshop, a typical data entry worker is interviewed and describes the following process:

> “I receive the paper invoices with the checks attached to them. I enter the invoice number into the system, and pull up the record of sale. I then enter the payment information into the system, including the payer name and address, the check number, date received, amount, checking account number and routing number.”

Brief Use Case:

> The System displays a screen that allows the User to enter an invoice number. The System accesses the sales record and displays the current information in the record to the User, highlighting fields that are related to payment. The User types information into the fields and submits the updated record. The System updates the sales record.

Example: Data Entry(Fully Dressed)

Use Case Name: Enter Payment Data Into Sales Record System

Scope: Sales Record System

Level: User-goal

Actors: Data Entry Worker(Primary), System (Supporting)

Stakeholders:

- Data Entry Worker: Enter payment information for invoice into system
- Bank: Properly track payments for sales

Preconditions: There is an active network connection between the Data Entry Worker and the Sales Record database

Example: Data Entry

Main Success Flow:

1. The use case begins when User enters an invoice number into a search screen
2. The System retrieves the Sales Record and displays the information to the User
3. The System highlights areas that are related to payment information
4. The User types the payment information into the payment fields indicated by the system
5. The User submits the record to the System to be updated

Example: Data Entry

Alternate flows (details left out):

Invalid Invoice Number

User attempts to submit the record to the System before all payment fields are filled in

User enters invalid information into payment fields

The System fails to update the record due to internal failure (database error)

Example: Data Entry

Post-Conditions (Success Guarantee):

Successful Completion

The sales record has been successfully updated in the sales database.

Failure Condition

The sales record has not been properly updated with the payment information in the sales database.

Special Requirements:

The System must contain a proper interface to an existing sales database.

The payment fields are dictated by the database design and invoice design

What is an Executive Summary (Brief)

- Very short (one page) summary, intended for high level executives
- Format:
  - Introductory paragraph, describes the purpose of the brief
  - Several bullet points that highlight the main message of the summary
  - Closing paragraph that sums it up
- Brevity is critical – keep this high level!
- Avoid typos, grammar mistakes, misspellings, etc.
  - Look professional

How to Find Use Cases

One the hardest, but most important parts of the projects

All understanding of the requirements, and hence system design, will flow from here

Interact closely with customer/client/user

- Here customer means the person purchasing the software

One great strategy – take an actor perspective, role play

- Each use case should be designed to satisfy a goal or a primary actor

The NextGen POS System

- Point-of-Sale (POS) system is application used to record sales and secure payment
  - Checkout line at store
- System includes hardware and software
- Has interfaces to various service apps, like tax calculator and inventory control, and the system should work even if access to these external services is down (i.e., at least allow checkout with cash if the credit card processing interface goes down)
- Needs to support multiple client-side interface types, like thin web-browser, touchscreen, wireless phone, etc.
- We plan to sell this to many types of businesses which may have different business processing rules – we need flexibility and the ability to customize

Defining Use Cases: System Boundaries

Choose a System Boundary

- Determine the edges of the system being designed – is it the software, the software and hardware? Does it include the person using the system?
- We are determining the Domain for the system

Think about our POS Case Study … what is the system boundary?

- In this case, the *system under design* is the software and hardware associated with it
- The cashier, store databases, payment authorization services, etc. are outside the boundary of the system
- Note that does not mean that they are not important – they are simply outside the boundary

Having trouble identifying the boundary? Identify the external actors, and that should help define it

Use Cases: Identify Primary Actors and Goals

Actors and goals are usually defined together, because in order to identify the primary actor, you usually must know what his/her goal is

Ask the fundamental questions: How is the system being used, and by whom?
- Who starts the action, who provides what, who monitors, etc.
- Remember, actors can be other systems, like a payment system or a database – not necessarily human users!

This is usually done in the requirements workshop brainstorm sessions

One useful approach: create a table, list all actors and goals, and then identify the primaries
- Remember, the primary actor is the one that has a need or goal fulfilled by the system
- Secondary (supporting) actors provide a service to the system
- Offstage actors have an interest in the behavior of the system, but are not primary or supporting

Note the primary actor may be defined based upon the choice of system boundary

Use Cases: Identify Actors, Goals

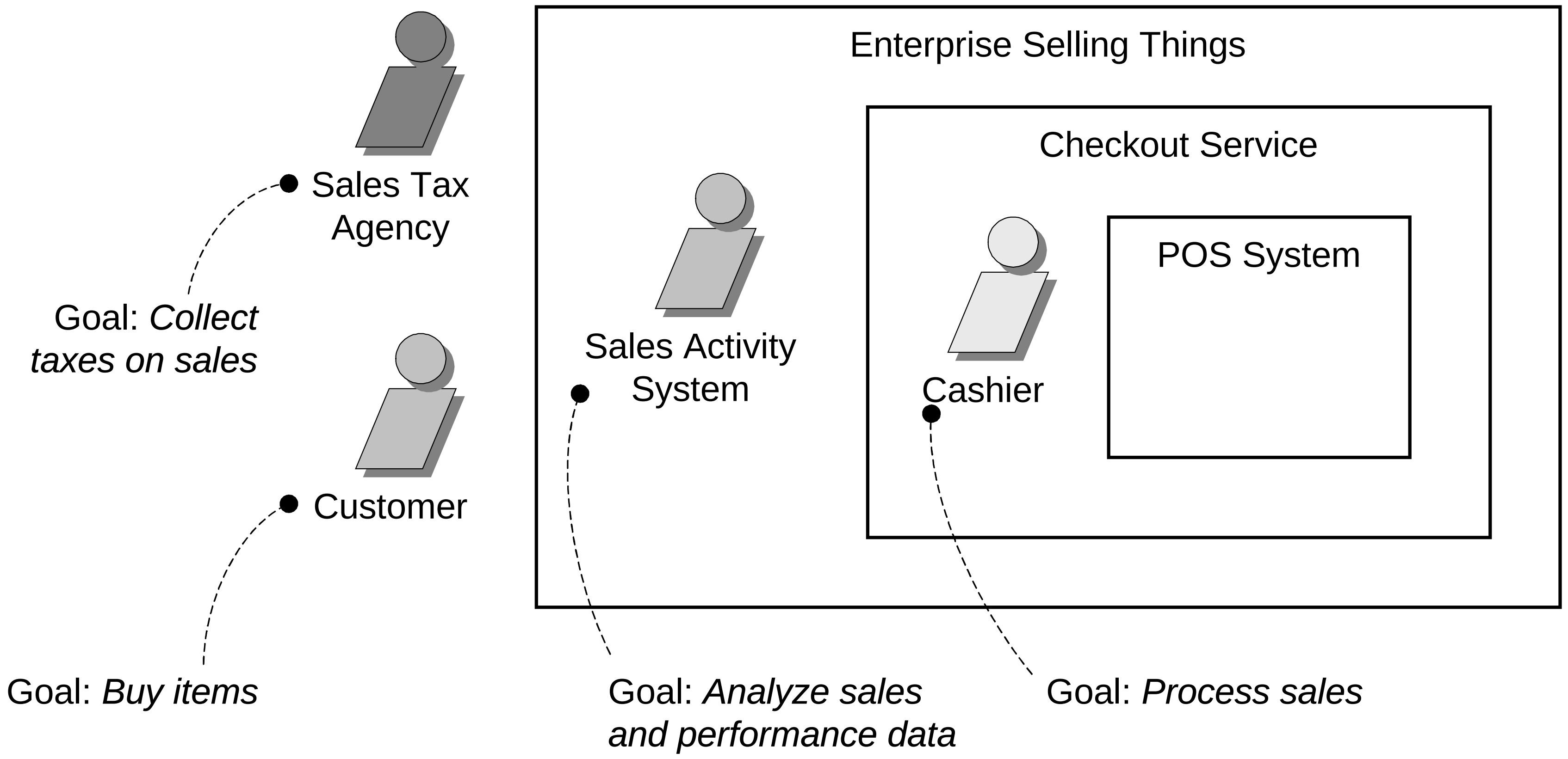
In Use Case Modeling, in order identify the actors and their goals, we need to understand the system from the actors’ perspectives

- Who is using the system? What are they doing with the system?

Best approach to this is to identify the actors and then ask “What are your goals? Are they measurable?”

- This gets to the heart of what the stakeholders really want from the system
- Also helps to avoid slipping into design too early – if we concentrate on the end goals, everyone stays open to new solutions to meet those goals

Use Cases: Primary Actors and System Boundaries



POS Case Study: Primary Actor

The previous diagram implies the cashier is the primary actor

What about the customer?

- Customer is an actor, but not primary. Why not?

Look at system boundary: For this case study, the system is the POS system. Unless self checkout, the customer’s goal is not the principle goal here

- The cashier’s goal (look up prices, get the total, process payment) is the main goal to be fulfilled by the system. **The system is being designed for use by the cashier – to meet his/her primary goal.**

Goes back to the basic domain definition – what are we designing?

Note: If the system under design is the entire sales enterprise system (inventory, sales tracking, POS), then the customer is the primary actor

Use Cases: Putting it all Together

First, define the system boundary. This involves defining the limits of what services the system will provide

Next, define the actors and their goals, including the primaries

- Role play, actor/goal tables
- Can also do this by analyzing events that may take place, and then identifying the actor generating the event and the goal of the actor

Finally, define the use case

- Generally do one use case for each user goal
- Use a name that describes the goal, start with a verb
- Often the CRUD goals are collapsed into one goal called *Manage X*, where X is whatever is being managed (e.g., “Manage User Accounts”)

Use Cases: How Big or Small?

It really depends on context

Generally, the use case should define a task performed by one actor in one place at one time, in response to an event.

If the use case is growing to many pages in length, consider creating sub-tasks

If the use case is only one sentence or step, probably too small and not worth exploring

Is there a basic business value in the use case?

- Which actors are impacted and how?
- What is the impact to the overall business?

Applying UML: Use Case Diagrams

Note that diagrams are secondary in use case development, and are often used just as an easy documentation tool. Use Cases are carefully crafted <u>text</u>.

Good to draw these when working out the actors and goals, and the actor-goal list

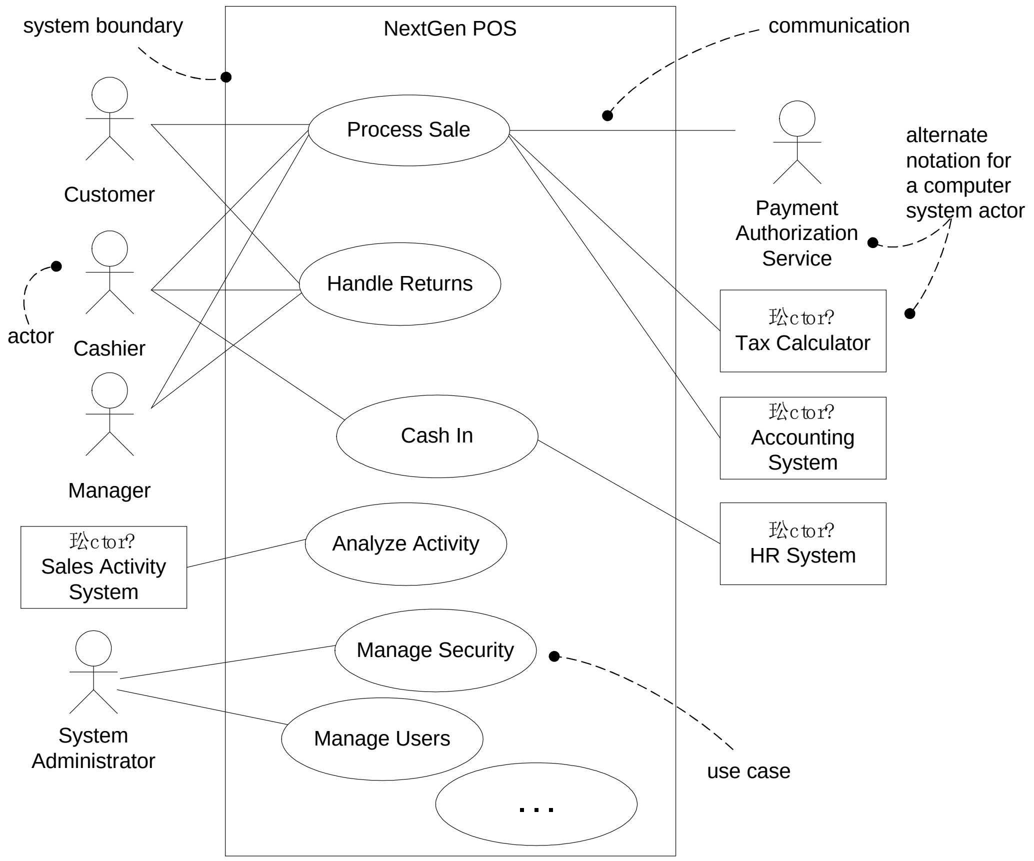
Can also be very helpful when deciding system boundaries – decide what is inside the system and what is outside

Use case diagrams are tools, and should be used to help understand the system and use cases under development

Never accept a set of use case diagrams as the "official" Use-Case Model for a project!

- They are much too vague, and you are leaving yourself open to misunderstanding later down the road
- Get it in writing!

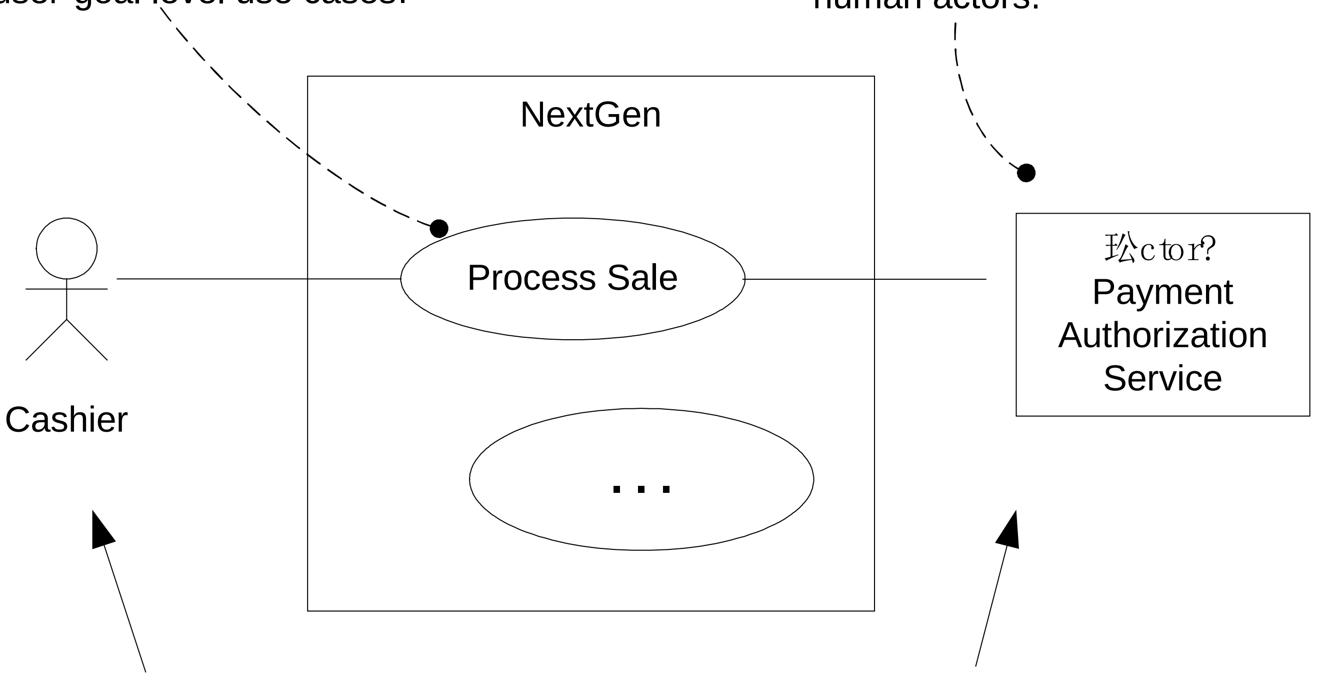
Example: POS


system boundary
NextGen POS
communication
Process Sale
Customer
Payment Authorization Service
alternate notation for a computer system actor
Handle Returns
actor
Cashier
«actor» Tax Calculator
«actor» Accounting System
Cash In
Manager
«actor» HR System
«actor» Sales Activity System
Analyze Activity
Manage Security
System Administrator
Manage Users
use case
. . .

UML Use Case Diagramming

For a use case context diagram, limit the use cases to user-goal level use cases.

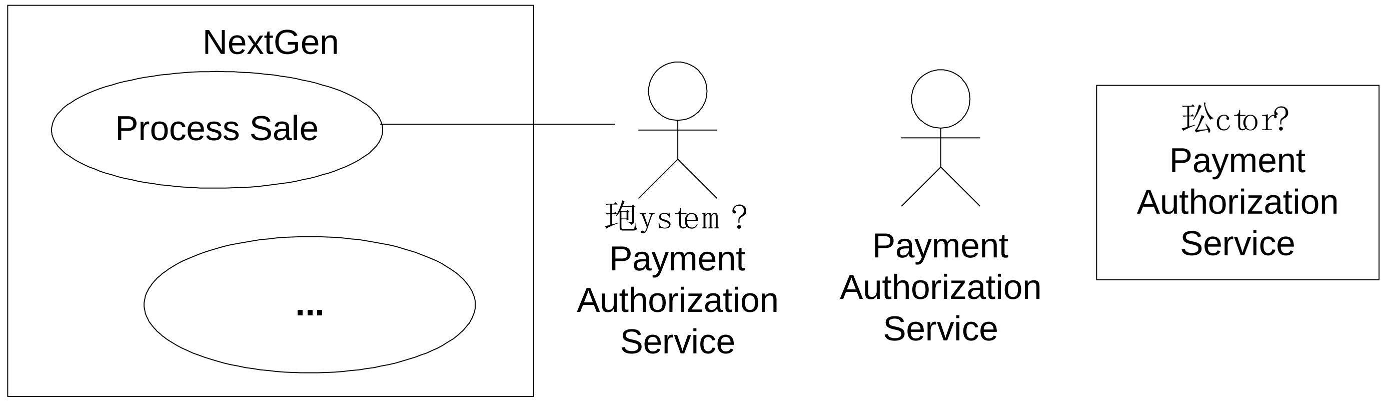
Show computer system actors with an alternate notation to human actors.



primary actors on the left

supporting actors on the right

UML Use Case Diagramming (alt)


NextGen
Process Sale
...
玸ystem？
Payment Authorization Service
Payment Authorization Service
玜ctor?
Payment Authorization Service

Some UML alternatives to illustrate external actors that are other computer systems.

The class box style can be used for any actor, computer or human. Using it for computer actors provides visual distinction.

Use Cases – Some Observations

Don’t design, don’t suggest coding. Think about the user story, the actors and goals

UML can also be used to create activity diagrams, which are workflow diagrams. We will discuss later

Avoid detailed function lists – this is old school, tedious, and not efficient. Write user stories.

- The Use-Case Model artifact captures the functional requirements

Remember, there are other UP artifacts that may still collect requirements: non-functional requirements, domain rules, etc. may be captured in the Supplementary Specification artifact.

The Vision document artifact *may* contain a list of high level system functions, as these are necessary to define the system scope

Case Two: The Monopoly Game

Software version of Monopoly game

This will run as a simulation; the user will configure the game (i.e. indicate the number of players, etc.), start the simulation, and let the game run to its conclusion.

A trace log will be created to record each move a player makes

Monopoly Game Use Case

This is an example of a project where most of the requirements are NOT captured by the use cases!

In this case, there is only one (simple) use case …

- The player starts the game (simulation)

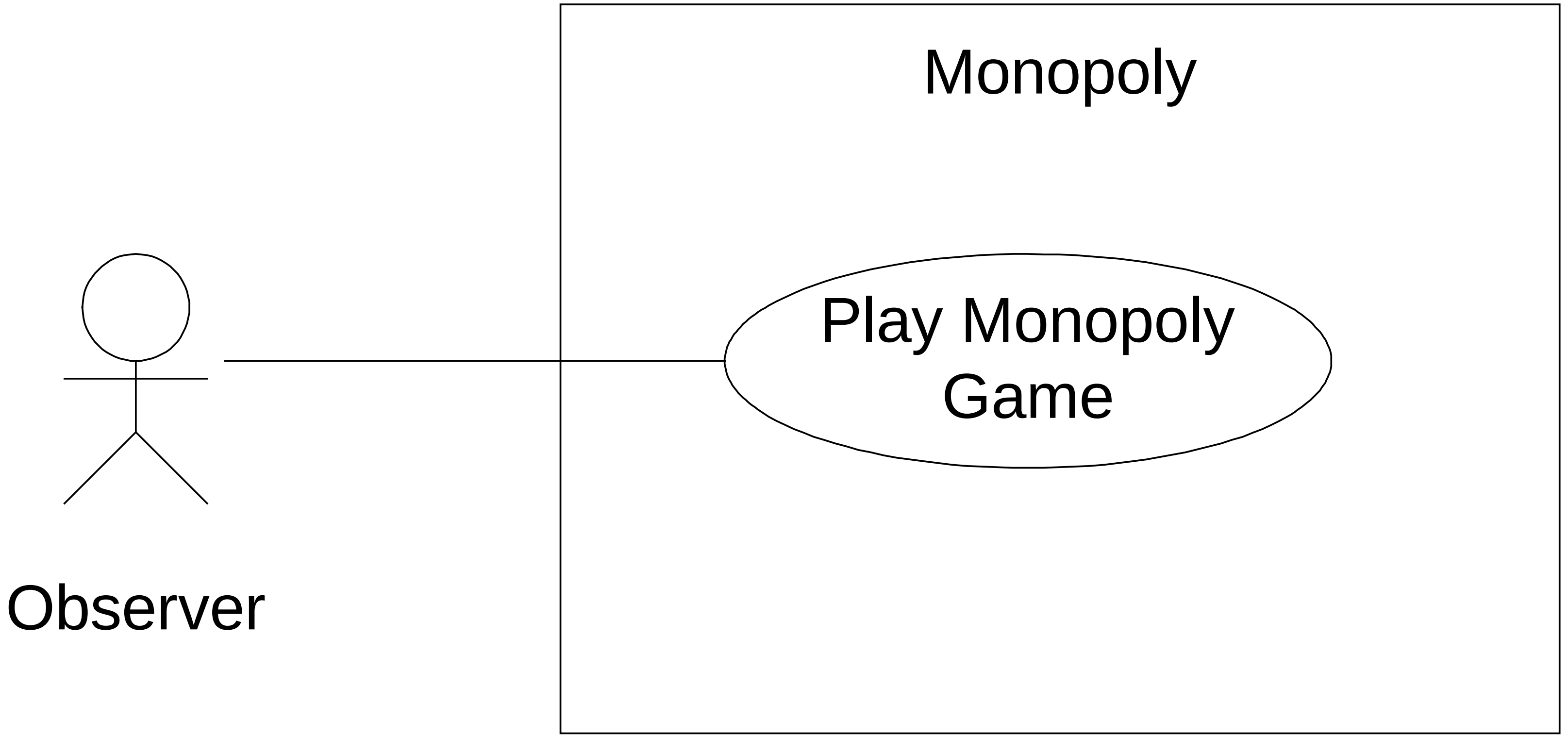
Are there no requirements?

- The *functional* requirements are simple, and basically captured in the project description on the last slide
- This system will have many *rules* requirements – namely, the rules of the Monopoly game.

These requirements, like most *business logic* requirements, would be captured in the Supplementary Specification (we will talk about this later).

- This type of requirement is not directly functional to the user, so usually not captured in the use case

Monopoly Game: Use Case Diagram



Monopoly Game Use Case

USE CASE UC1: Play Monopoly Game

Scope: Monopoly application

Primary Actor: Observer

Stakeholders and Interests: Observer: Wants to easily observe the output of the game simulation

Main Case Scenario:

1. Observer requests new game initialization, enters number of players
2. Observer starts play.
3. System displays game trace for next player move (see domain rules, and “game trace” in glossary for trace details)
4. Repeat step 3 until a winner is decided or the Observer cancels the simulation

Monopoly Game Use Case

USE CASE UC1: Play Monopoly Game (cont.)

Extensions:

*a. At any time, System fails:

(to support recovery, System logs after each completed move)

1. Observer restarts System.
2. System detects prior failure, reconstructs state, and prompts to continue.
3. Observer chooses to continue (from last completed player turn).

Special Requirements:

- Provide both graphical and text trace modes.

Use Cases and Iterative Methods

UP is use-case driven development

Functional requirements are primarily captured in the use cases (i.e. the Use-Case Model) – this means you would not expect to find many functional requirements elsewhere

Critical to planning iterations

- Planning usually involves selecting a use case to work on, or enhancing an existing use case

In UP, the main goal for each team is to design objects which, when collaborating together, will *realize* a use case. This is true of Agile methods in general.

Use cases often give valuable material for the creation of user manuals

Use cases also provide valuable ideas for testing

Use cases often developed in Requirements Workshops held with the customer

Iterative Development: Disciplines, Phases, and Artifacts

Discipline	Artifact	Inception	Elaboration	Construction	Transition
Business Modeling	Domain Model		S, R		
Requirements	**Use-Case Model** Vision Supp Spec Glossary	S S S s	R R R R		
Design	Design Model SW Arch Doc		S S	R	

S – Start, R - Refine

Relating Use Cases

Often we need to relate one use case to another, such as the withdrawing money from the ATM and establishing a session on the ATM use cases

Note that this does not change the behavior of the system – it’s simply a way to better organize a set of use cases that hopefully makes the system more understandable

- Also reduces redundant text, which makes it easier to manage documentation

Relating Use Cases

These relationships are tools that can help organize use cases – they should not be the focus of the use case development effort

- Spend your time *writing text*, not debating diagrams and relationships

Keep in mind, the organization of use cases by relationships may evolve over the iterations (in the Elaboration Phase, for example). It is not worth the effort to try to get all the relationships defined up front – this is Waterfall thinking

Terminology: Types of Use Cases

A *concrete* use case is initiated by an actor and performs the entire behavior desired by the actor to meet a goal.

- These are the elementary use cases that we have seen earlier

An *abstract* use case is a use case that is never instantiated on its own; it is always a part of another use case, like a “subfunction” use case.

- For example, in our POS case study, we may have a *Process Sale* use case, and a part of that use case may be *Handle Credit Payment*. The first use case is concrete, the second is abstract, because it is always used as part of another use case.

Terminology: Types of Use Cases

Abstract use cases often evolve later in the iteration process, when it is noticed that certain process steps are repeated in multiple use cases.

Fundamental process in developing reusable code: If you see commonly occurring themes (code), abstract out and create sub-classes

The use case that includes another use case (or is extended or specialized by another use case) is called the *base use case.* The use case that is the inclusion, extension, or specialization is called the *addition use case.*

Relation between use cases

Use cases are business processes and

- A business process **may** (conditionally) ***include*** another business process.

- The execution of one business process **may** continue with the execution of another business process.

The *include* Relationship

Most common and important

This will involve an abstract use case that contains some behavior that is common to several other concrete use cases

If we consider the previous example, we may have something like this in the Process Sale use case:

Main Success Scenario

1. Customer arrives at POS system with goods/services to purchase

…

7. Customer pays and System handles payment

Extensions:

7a. Paying by credit: Include *Handle Credit Payment*

7b. Paying by check: Include *Handle Check Payment*

The *include* Relationship

The *Handle Credit Payment* use case would then have its own use case description, complete with Main Success Scenario and Extensions

Use of the “Include” keyword is optional; it can be left off

When using included use cases, hyperlinks (either in document or on line) is very useful

This can also be used to break up a very complex use case into higher level steps which are then detailed in the included use cases

This can also be used to capture branching behavior or asynchronous behavior in the use case

These are actions that the user may take at any time in main success scenario, and thus are not associated with any particular step

The *include* Relationship

To include an asynchronous branch point in a use case, we usually use the following notation:

Extensions

*a. At any time, the Customer selects to cancel the request: *Cancel Request*

*b. At any time, the Customer selects to return to main menu: *Return to Main Menu*

3 – 10: Customer requests help: *Show Help Screen*

Note the last extension indicated that this included use case can be instantiated at any time between steps 3 and 10 (inclusive)

The *extend* Relationship

This relationship is used to add to a use case that has been more or less frozen for some reason

- Sometimes expanding or extending a use case is troublesome, and needs to be avoided

Use “Extension Points” to extend the base use case, and then write the subfunction use case separately

Process Sale (Base Use Case)
…
Extension Points: *Payment in Step 7*

…

7. Customer pays and System handles payment

The *extend* Relationship

Pay With Gift Certificate (Extended Use Case)
…

Trigger: Customer wants to use a gift certificate

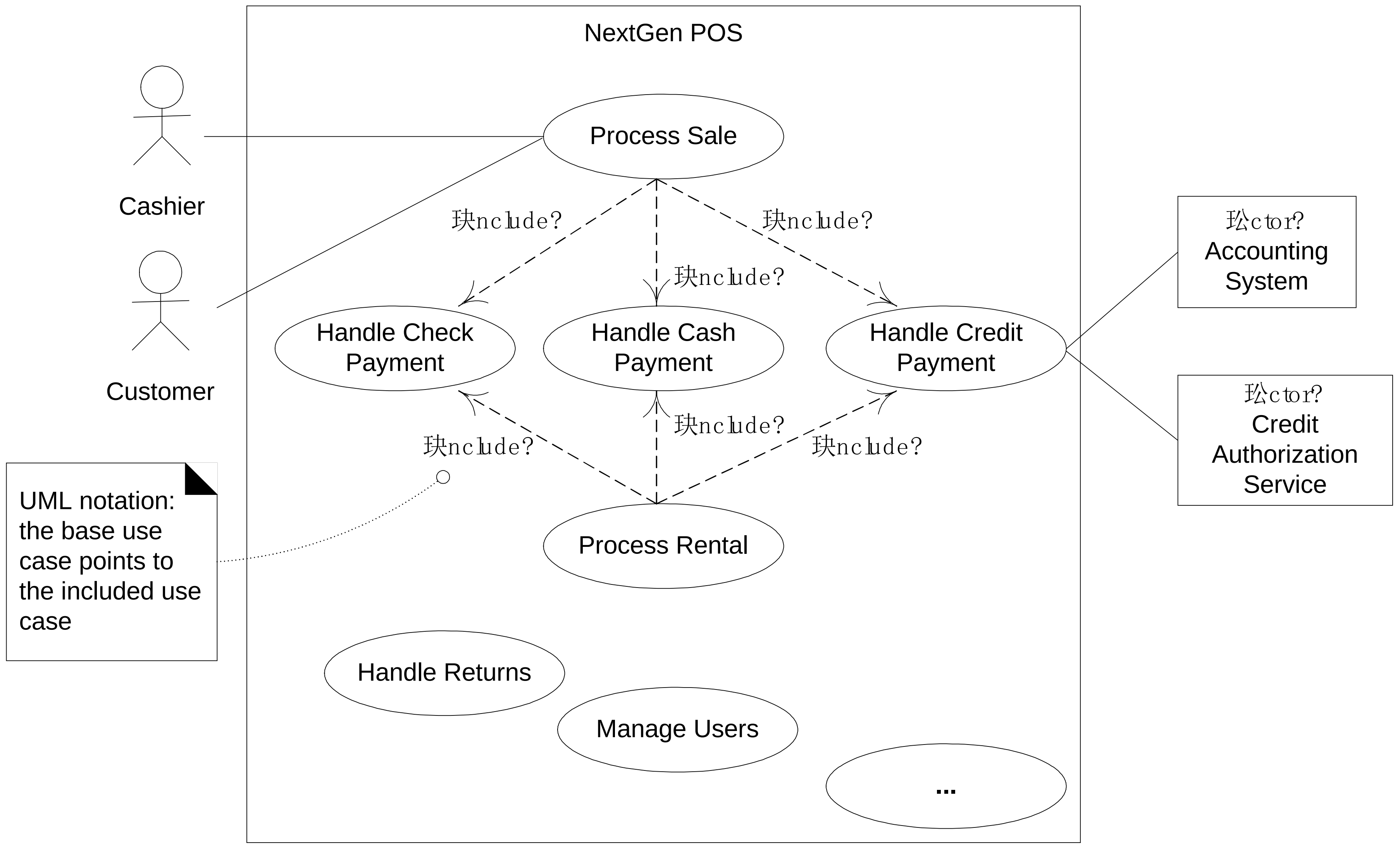
Extension Points: Payment in Process Sale

Level: Subfunction

Main Success Scenario:

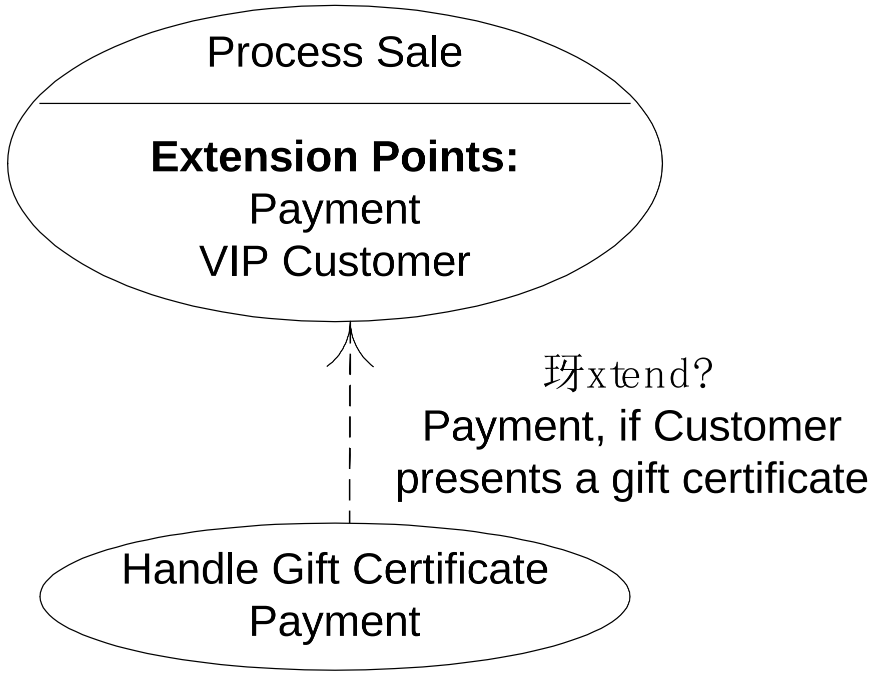
1. Customer gives gift certificate to the Cashier
2. …

Note the extended use case refers to the Extension Point, not a numbered step in the base use case - more robust.



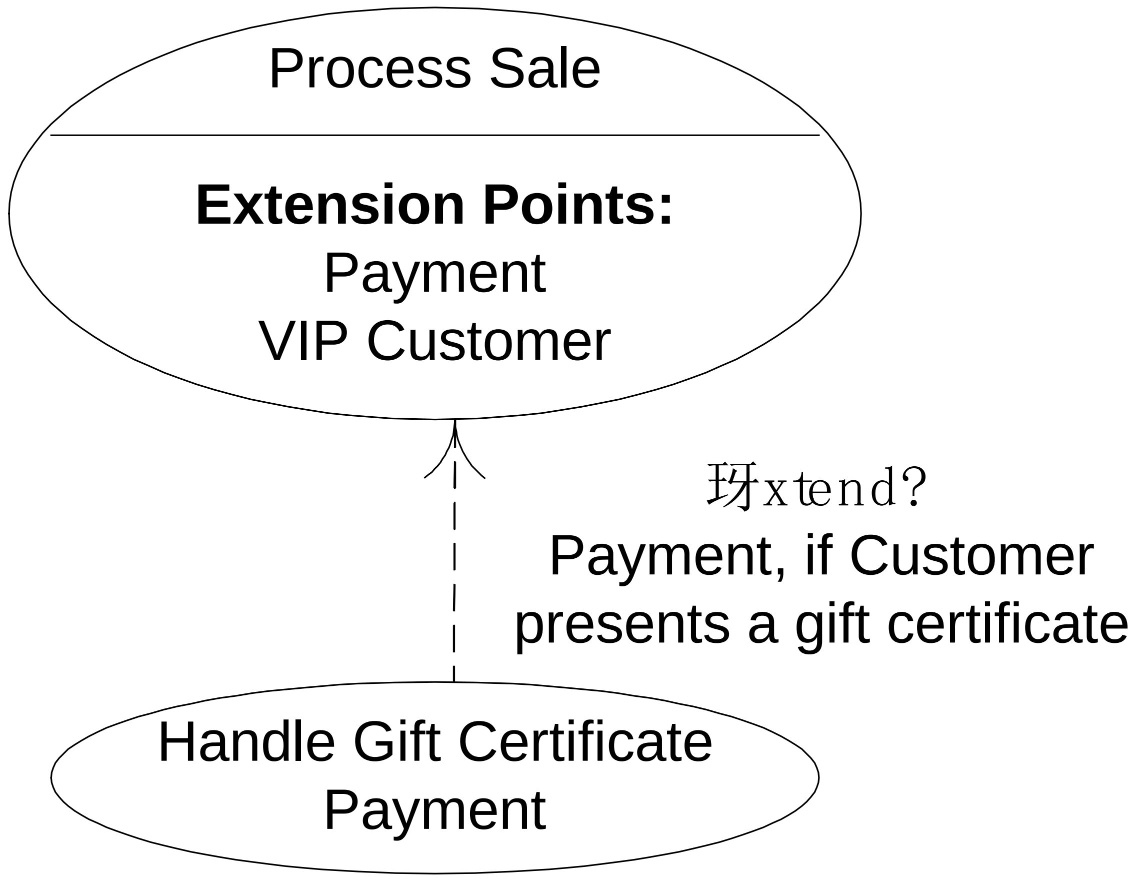
This diagram shows how to indicate an “include” relationship in a UML diagram

UML Diagrams – Extend Relationship


Process Sale
Extension Points:
Payment
VIP Customer
玡xtend?
Payment, if Customer presents a gift certificate
Handle Gift Certificate Payment

UML notation:
1. The extending use case points to the base use case.

2. The condition and extension point can be shown on the line.


Process Sale
Extension Points:
Payment
VIP Customer
玡xtend?
Payment, if Customer presents a gift certificate
Handle Gift Certificate Payment

UML notation:
1. The extending use case points to the base use case.

2. The condition and extension point can be shown on the line.

This diagram shows how to indicate an “extend” relationship in a UML diagram

Other Requirements

In the UP, there are some other important requirements artifacts other than the Use-Case Model

- These may not be as important to OOA/OOD, but they are needed for the project to succeed

Other important artifacts that capture requirements:

- Supplementary Specification: Most non-functional requirements, like reporting, packaging, documentation, etc
- Glossary: Terms and Definitions
- Vision: High level system description
- Business Rules: Rules that may transcend the current project, i.e. tax laws, etc.

Supplementary Specification

- Look for requirements, information, and constraints not easily captured in the use cases or Glossary
- Use FURPS+
  - In particular, look for Usability, Reliability, Performance, and Supportability requirements
  - These are known as the “Quality Requirements”
- This will be important when we get to architecting the system
  - “architectural analysis and design are largely concerned with the identification and resolution of the quality attributes in the context of the functional requirements”
  - In other words, we want to design the system to meet the functional requirements for the user while still meeting the quality requirements
- Note the Supplementary Specification may also contain business logic rules, like application-specific calculations.

System Sequence diagrams

Creating SSDs

Identify a particular course of events to demonstrate

- This is usually found in a use case – each SSD demonstrates a particular use case main success scenario

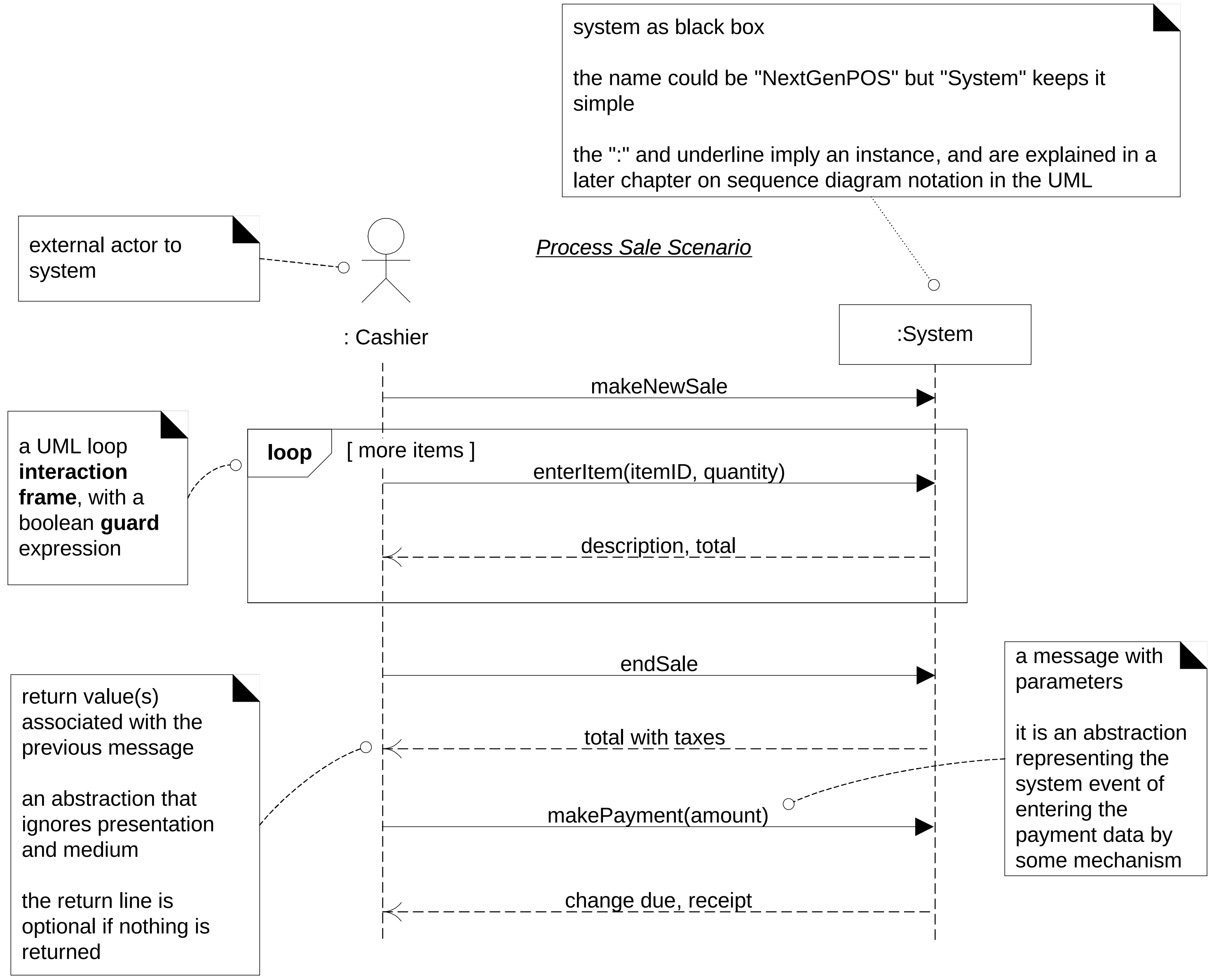
The SSD will show the external actors that interact with the system, the system itself (as a black box), and the system events that the actors generate

Note this is a simple UML Interaction Diagram – at this level (Use-Case Model), we do not have much information on the design of the system we are creating

- We are simply trying to identify the system events; we will later define object operations from here

Next slide shows an SSD for the cash-only Process Sale scenario in the NextGen POS project.

Note the use of the UML interaction diagram notation we discussed last time


system as black box

the name could be "NextGenPOS" but "System" keeps it simple

the ":" and underline imply an instance, and are explained in a later chapter on sequence diagram notation in the UML
external actor to system
Process Sale Scenario
: Cashier
:System
makeNewSale
a UML loop **interaction frame**, with a boolean **guard** expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale
a message with parameters

it is an abstraction representing the system event of entering the payment data by some mechanism
return value(s) associated with the previous message

an abstraction that ignores presentation and medium

the return line is optional if nothing is returned
total with taxes
makePayment(amount)
change due, receipt

SSDs – Why?

The SSDs are useful for capturing overall system *behavior*

- What events the system is expected to handle, what response (if any) is given to the user (primary actor)

Note that this is still very much an analysis tool - it is explaining what they system is expected to do, not *how* it is to be done

- This is why the system is modeled as a black box

Later, in the Design Model, we will explore how the system will handle the events it is presented with

- This will include more detailed inter-action diagrams (like we saw last lecture) showing how the software classes interact to handle the system events

Also useful for capturing interactions with other (external) systems, i.e. “supporting actors”

Creating SSDs … Key Ideas

Create an SSD for *one scenario* of a use case

Try to name the system events at the abstract level of intention, rather than in terms of a system device

- So “enterItem” is better than “scan”, as the latter implies some type of scanner

Usually best to start the event names with a verb if possible – implies action

SSDs sometimes will include interactions with external systems – we will see more later

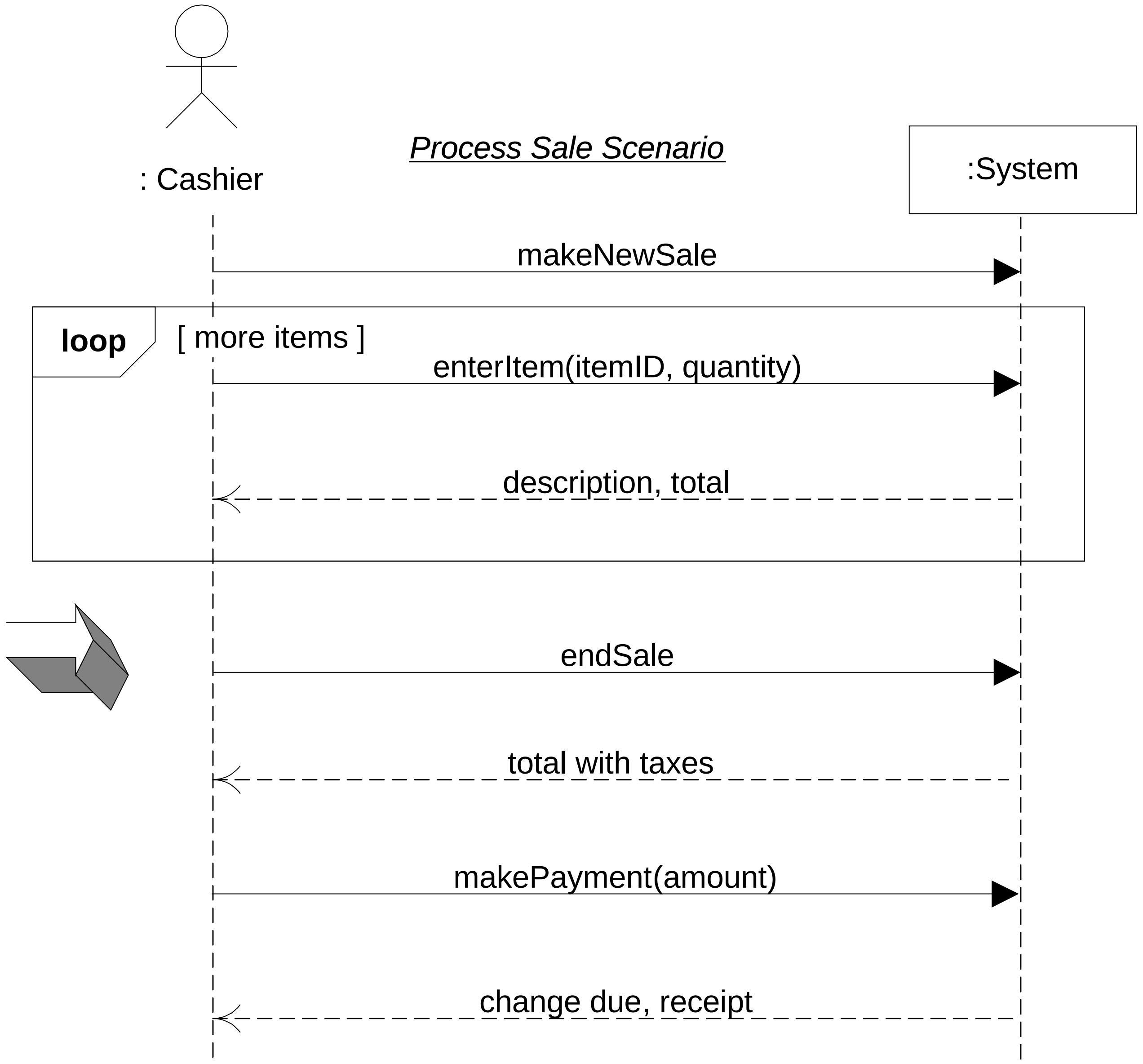
Often, terms and expressions included in the SSD are added to the Glossary while the SSDs are being built

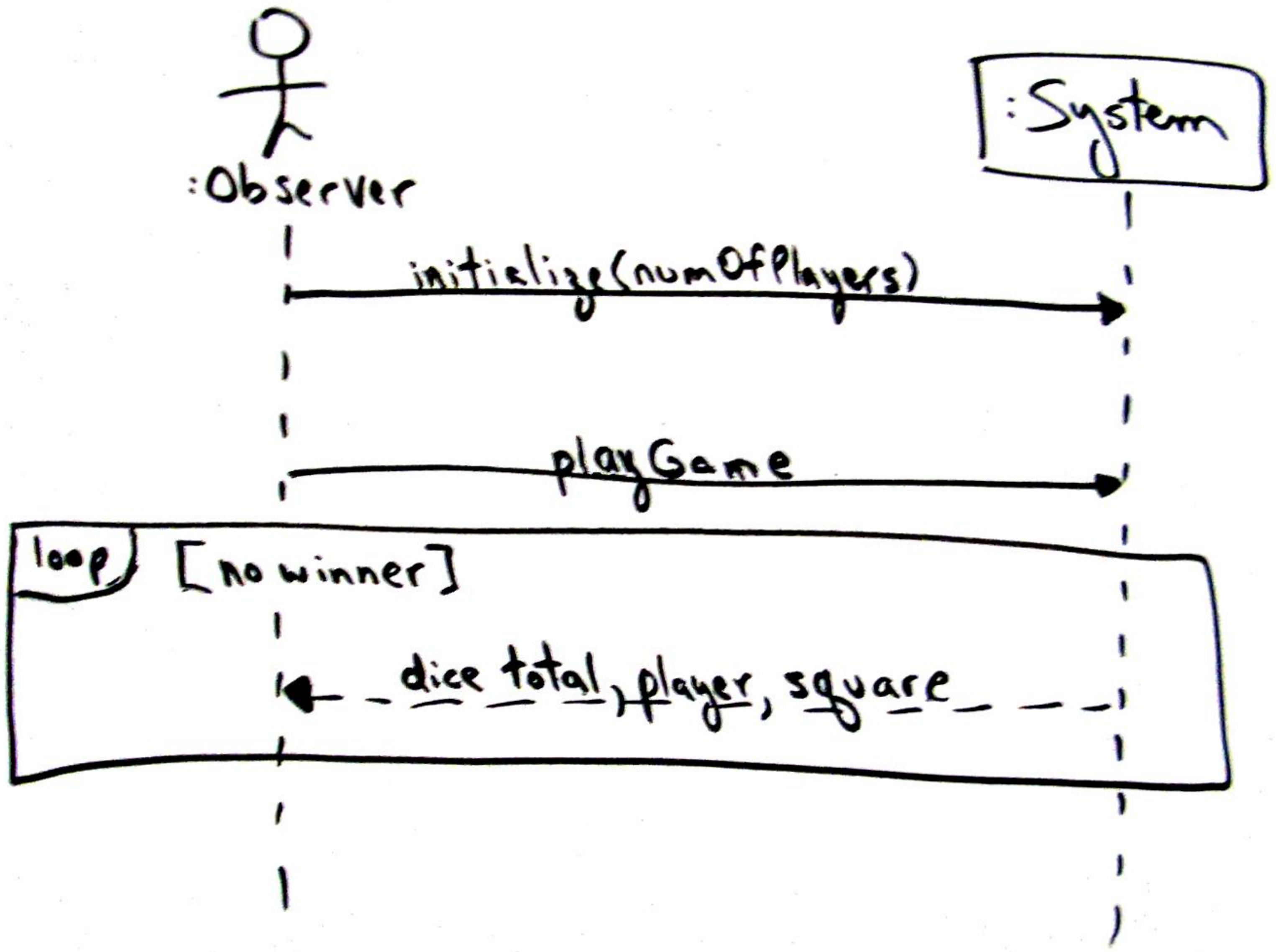
These are iterative, just like the Use-Case Model artifact that they belong to

- Add more scenarios as new SSDS as the project progresses
- May add details to an existing SSD, or use an existing SSD as a part of a new 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.
...




:Observer
:System
initialize(numOfPlayers)
playGame
loop
[no winner]
dice total, player, square

intro to domain models

Case Study: NextGen POS Requirements (Inception)

Key Decisions made during Inception, plans for first iteration:

- Implement basic, key scenario of Process Sale use case – entering items and receiving cash payment
- Implement Start Up use case to initialize the system
- No collaboration with external services at this point
- No complex pricing rules
- Notice that for the first iteration, the use case is limited – it will be expanded in later iterations

Case Study: Monopoly Requirements (Inception)

Implement basic, key scenario of Play Monopoly Game use case:

- 2-8 players
- Play for 20 rounds. Each round, each player takes one turn. During a turn, the player advances his/her piece clockwise around the board a number of squares equal to the number rolled on two six-sided dice.
- The results of each player’s roll is displayed: The display includes the name of the player and the value of the roll. When a player lands on a square, the player’s name and the name of the square are displayed
- There is no money, winner, or loser, no properties to buy or rent, no special squares
- There are 40 squares on the board: One is named “Go”, the others are simply numbered as “Square 1”, “Square 2”, … “Square 39”
- The only user input required is the number of players

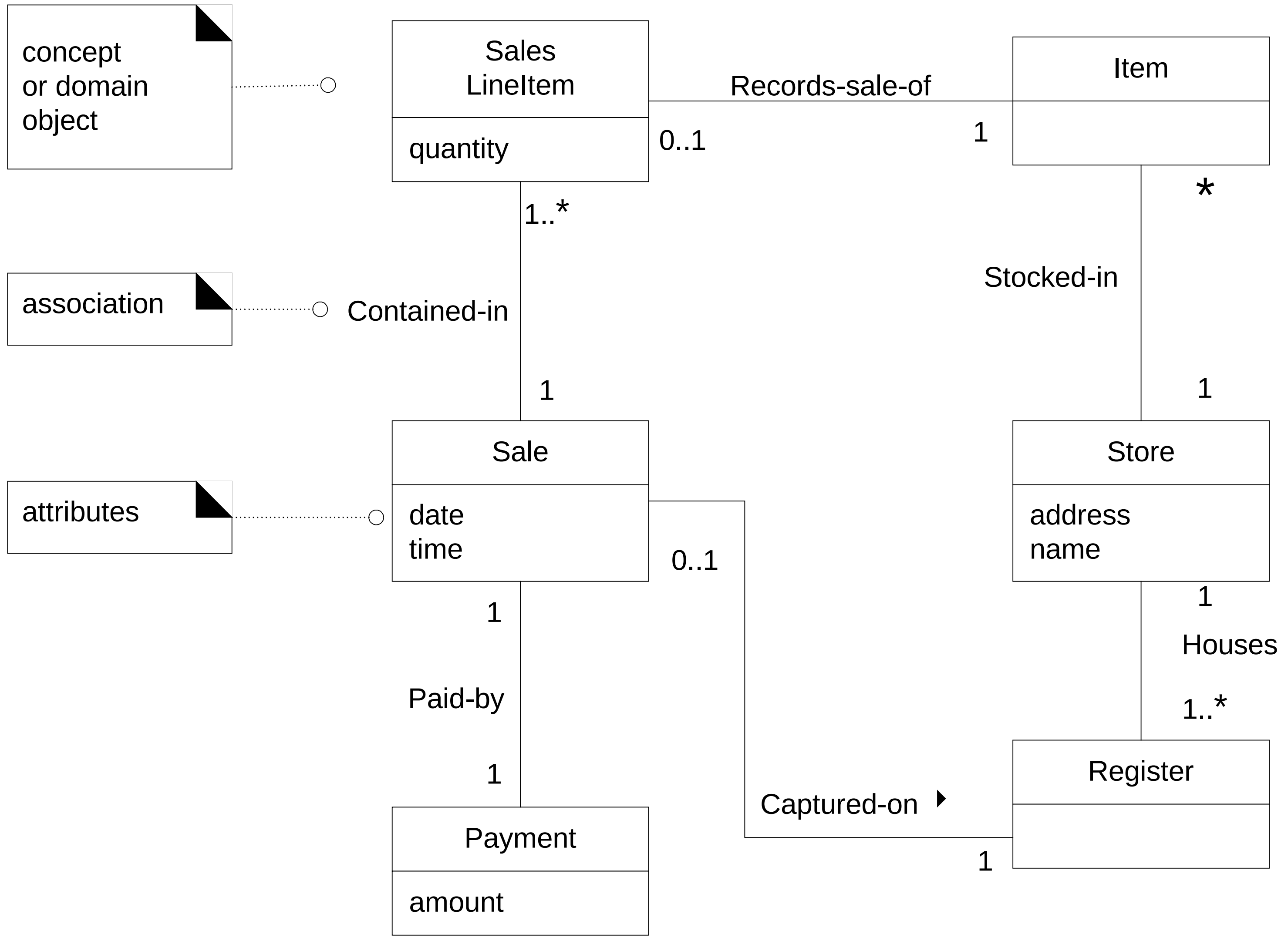
Domain Model - Introduction

Very important model in OOA … started after some key use cases have been developed

Illustrates the important concepts in the Domain, and will inspire the design of some software objects

Also provides input to other artifacts

- Use-Case Model
- Glossary
- Design Model (Sequence Diagrams)



This diagram shows an example of a an early Domain Model for the POS system.

Note …

We will be using the UML diagrams we saw earlier

The Domain Model appears to capture the important **conceptual classes** that make up the domain

- We will see how to identify these shortly

The model also shows how these classes inter-relate to each other

Key aspect of OOA: Identifying a rich set of conceptual classes

Remember, this is an iterative approach – don’t need the entire Domain Model created at once!

- The model is refined over the iterations in the Elaboration Phase

Domain Model: Definition

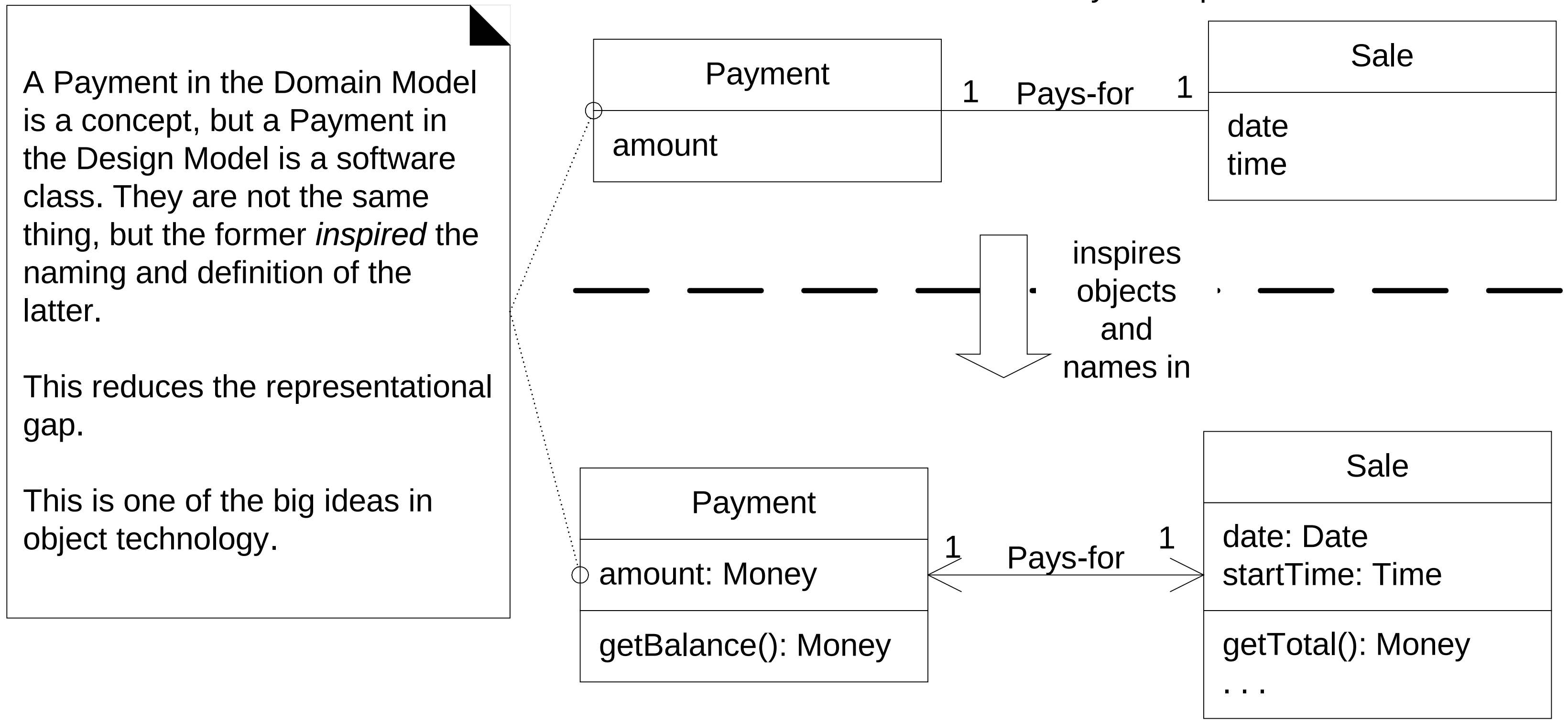
The Domain Model can be thought of as a *visual* representation of conceptual classes or real-situation objects in the domain (i.e. the real world).

In UP, the term Domain Model means a representation of real-situation conceptual classes, not software objects. The term does not mean a set of diagrams describing software classes, the domain layer of the software architecture, or software objects with responsibilities

This artifact is created in the Business Modeling discipline

Think of as a visual dictionary describing the domain: important abstractions, domain vocabulary, and information content

UP Domain Model
Stakeholder's view of the noteworthy concepts in the domain.



A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

UP Design Model
The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

The difference between domain model and design model – UML used in two different ways.

Creating Domain Models

This is dependent upon which iteration cycle you are in, but in general there are three steps:

1. Find the conceptual classes
2. Draw the classes as UML diagrams (conceptual level)
3. Add associations and attributes

Finding Conceptual Classes

- Use or modify existing models – we will see some of these later
- Use a category list
- Identify noun phrases in the use cases

Category Lists

This is a list of common conceptual class categories, generalized to apply to many situations

Can be used as a starting point; look for these conceptual classes in your domain

- Book has good list …
- Business transactions, transaction line items, where is the transaction recorded, physical objects, catalogs, other collaborating systems, ..

You can make a list of categories (or use a pre-existing list), and after reviewing use cases and requirements, list all conceptual classes you find that relate to a particular category

Noun Phrase Identification

Look at a textual description of the domain, and identify all the nouns and noun phrases

- Try not to do this mechanically – not all nouns are conceptual classes!

Good place to start is the fully dressed use case

- Go through the main success scenario, identify all important nouns, use these to name conceptual classes

Example: POS Use Case

Main Success Scenario (cash only):

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

(repeat 2-3 until no more items)

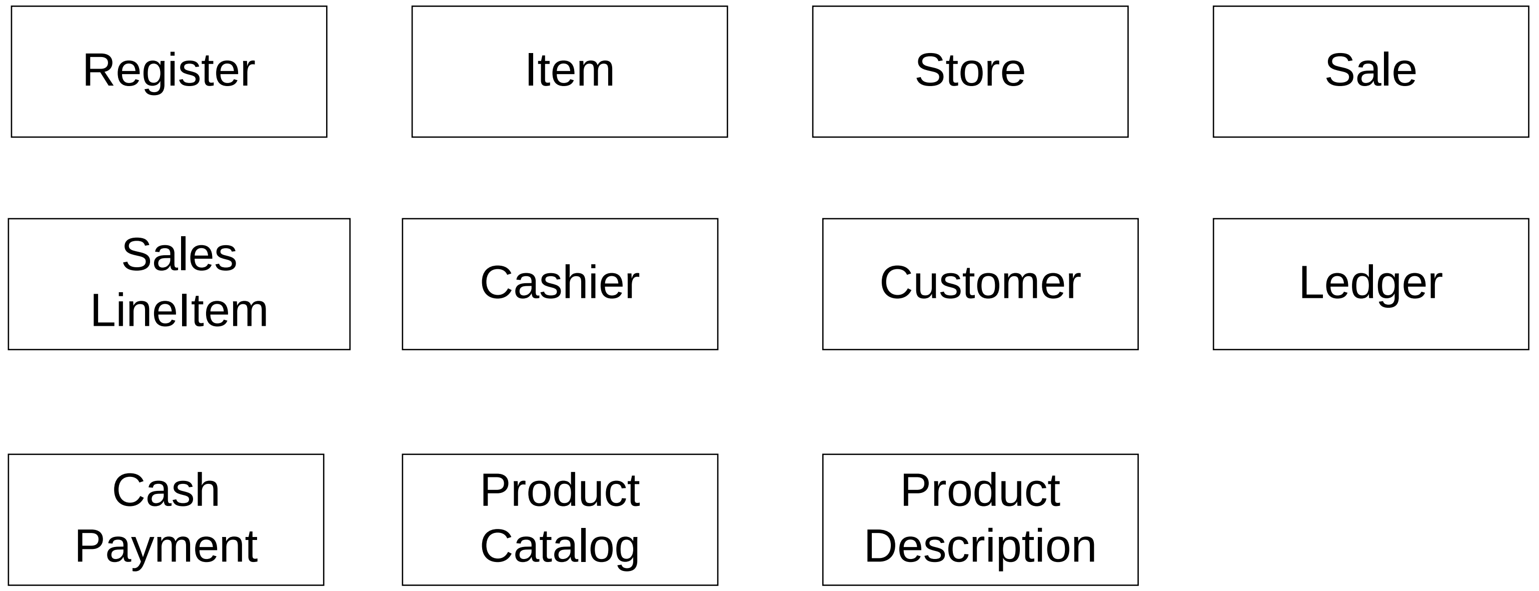
Example: POS Use Case (identify key nouns)

Main Success Scenario (cash only):

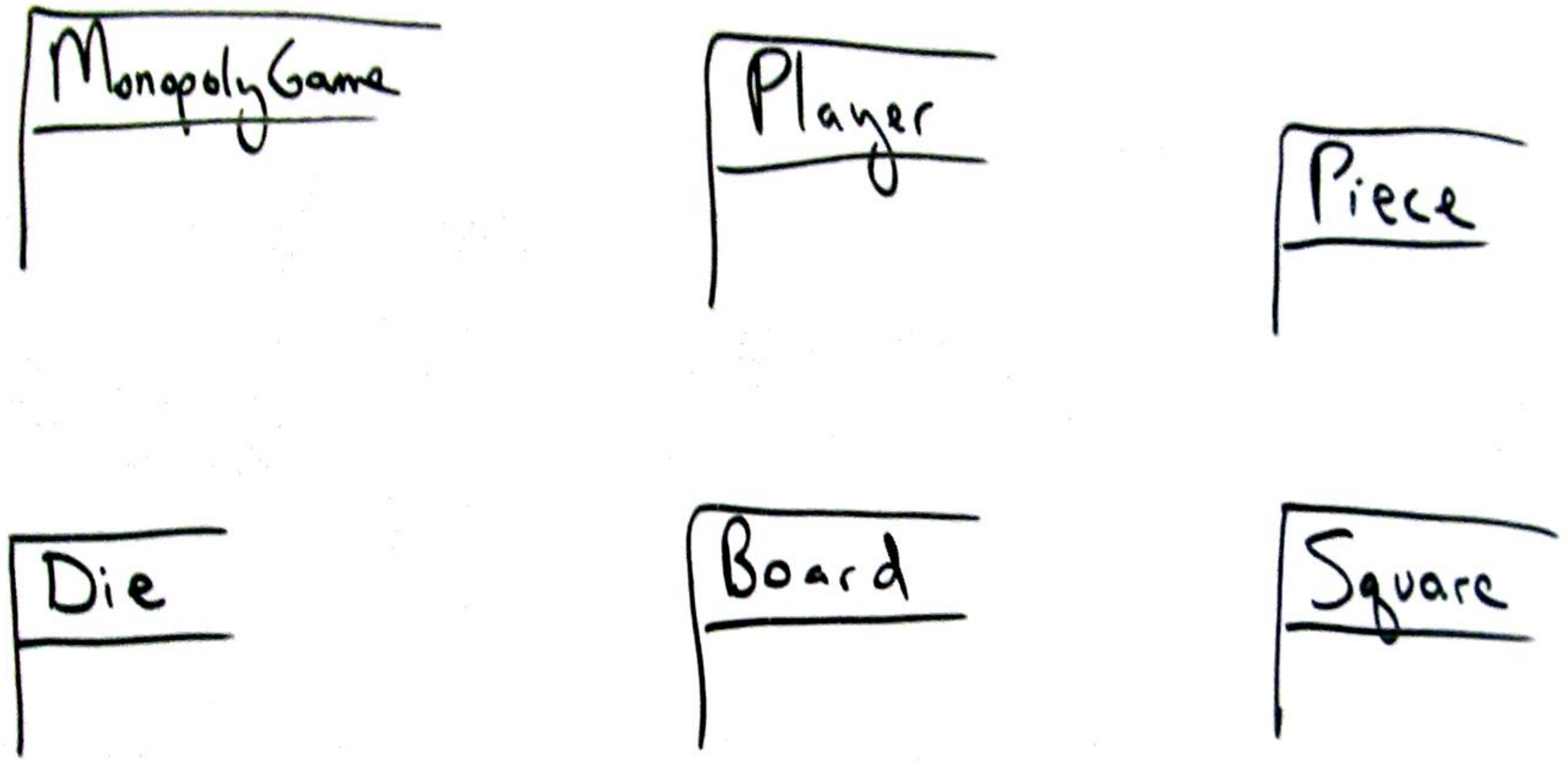
1. **Customer** arrives at **POS checkout** with **goods** and/or **services** to purchases
2. **Cashier** starts new **sale**
3. Cashier enters **item identifier**
4. System records **sale line item** and presents **item description**, **price**, and running **total**

(repeat 2-3 until no more items)

Example – Initial Draft of Domain Model for POS



Example – Initial Draft of Domain Model for Monopoly



Observations

This model will evolve as the project goes through iterations

But aside from that, why save this model? Once it has served its purpose, it can be discarded

- Once the more detailed class diagrams are created, there may not be a need for this model

It can be maintained in a UML CASE tool (there are many available)

Observations

Note not all conceptual classes need to be included – we are not going for a complete model in the beginning

- Be careful with classes that simply report information derived from other classes – like *Receipt*.
- If the reporting entity has some importance in the use case being considered (*Receipt* would be useful for *Handle Returns*, for example), then it should be included

Guidelines

Think like a mapmaker

- Use existing names you find in the requirements/use cases, don’t invent new ones
- Use terminology that is consistent with the business area

Exclude irrelevant or out of scope features

- For example, in the Monopoly first iteration, we are not using “cards”, so they do not need to be modeled
- Likewise, for the NextGen POS system, we do not need to include tax rules yet

Guidelines

Never add a conceptual class for something that is not there!

- Always model the real system – don’t try to design ahead
- Remember – think like a map-maker

Don’t be afraid of abstract conceptual classes

- Virtual connection, etc.
- If these are important to the real-world system, they should be modeled in the Domain model

Attributes and Conceptual Classes

Be careful not to turn conceptual classes into attributes

- If X cannot be thought of as a number or text, it is probably a conceptual class

For example, in the POS case study, the *Store* is not a number or some text, so it should be modeled as a conceptual class (and not an attribute of Sale, for example)

In Monopoly, the Piece, Board, Square, and Dice are not numbers or text, so they will be conceptual classes

The number that each dice rolls, however, can be thought of as an attribute

Description Classes

Often it is a good idea to include the information that describes a class (conceptual or software) in a separate class, called a *description class* (also called a *specification*).

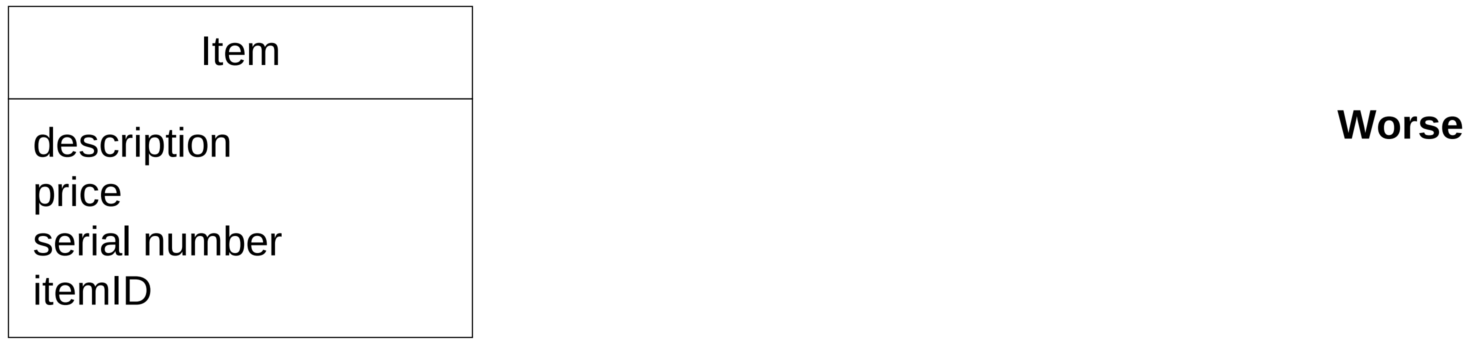
This is a more robust way to design the conceptual classes

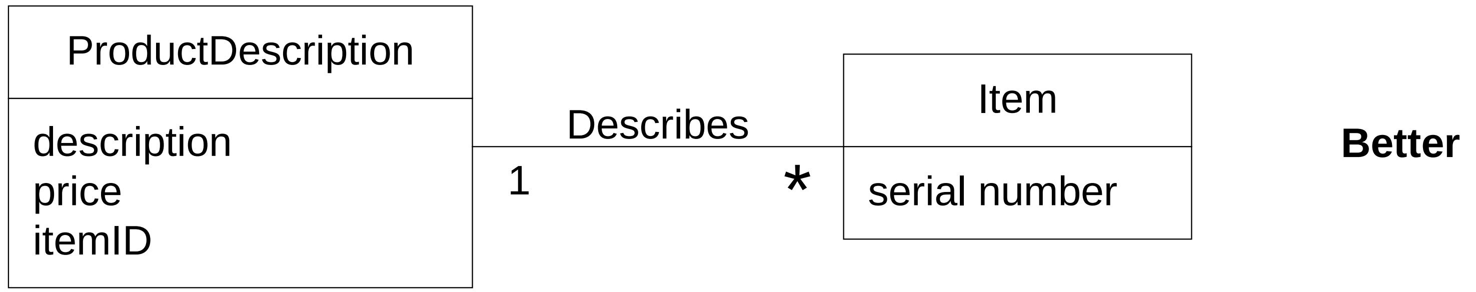
- Putting all information in each instance is wasteful because it duplicates information, and may be error-prone (what if something changes?)

Common in sales, product, and service domains.

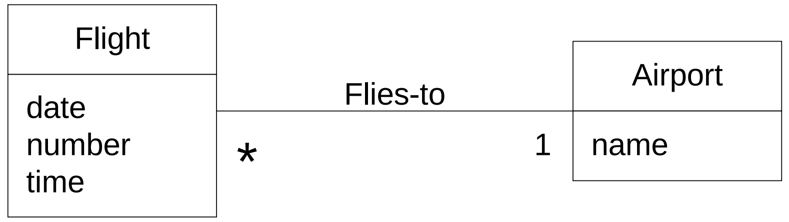
These are separate objects (conceptual classes, or real software classes)

Descriptor Class – Store Item

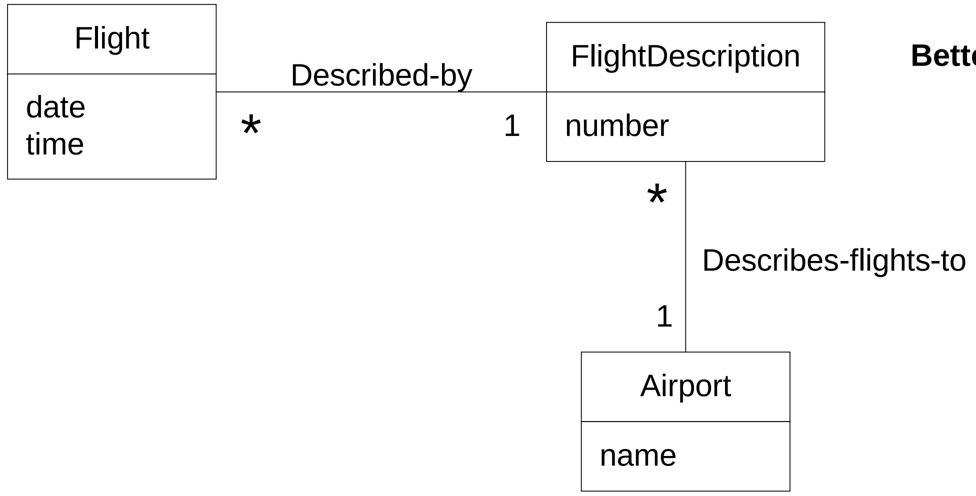




Descriptor Class – Airline Flight



Worse



Better

Associations

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

When to add an association between conceptual classes to the domain model?

- Ask “do we require some *memory* of the relationship between these classes?”
- The knowledge of the relationship needs to be preserved for some duration
- For example, we need to know that a *SalesLineItem* is associated with a *Sale*, because otherwise we would not be able to do much with the *Sale* (like compute the total amount, print receipt, etc.)
- For the Monopoly example, the *Square* would not need to know the value of the *Dice* roll that landed a piece on that square – these classes are probably not associated

Associations

Avoid adding too many associations

- A graph with *n* nodes can have *(n x (n – 1)/2)* associations, so 20 classes can generate 190 associations!

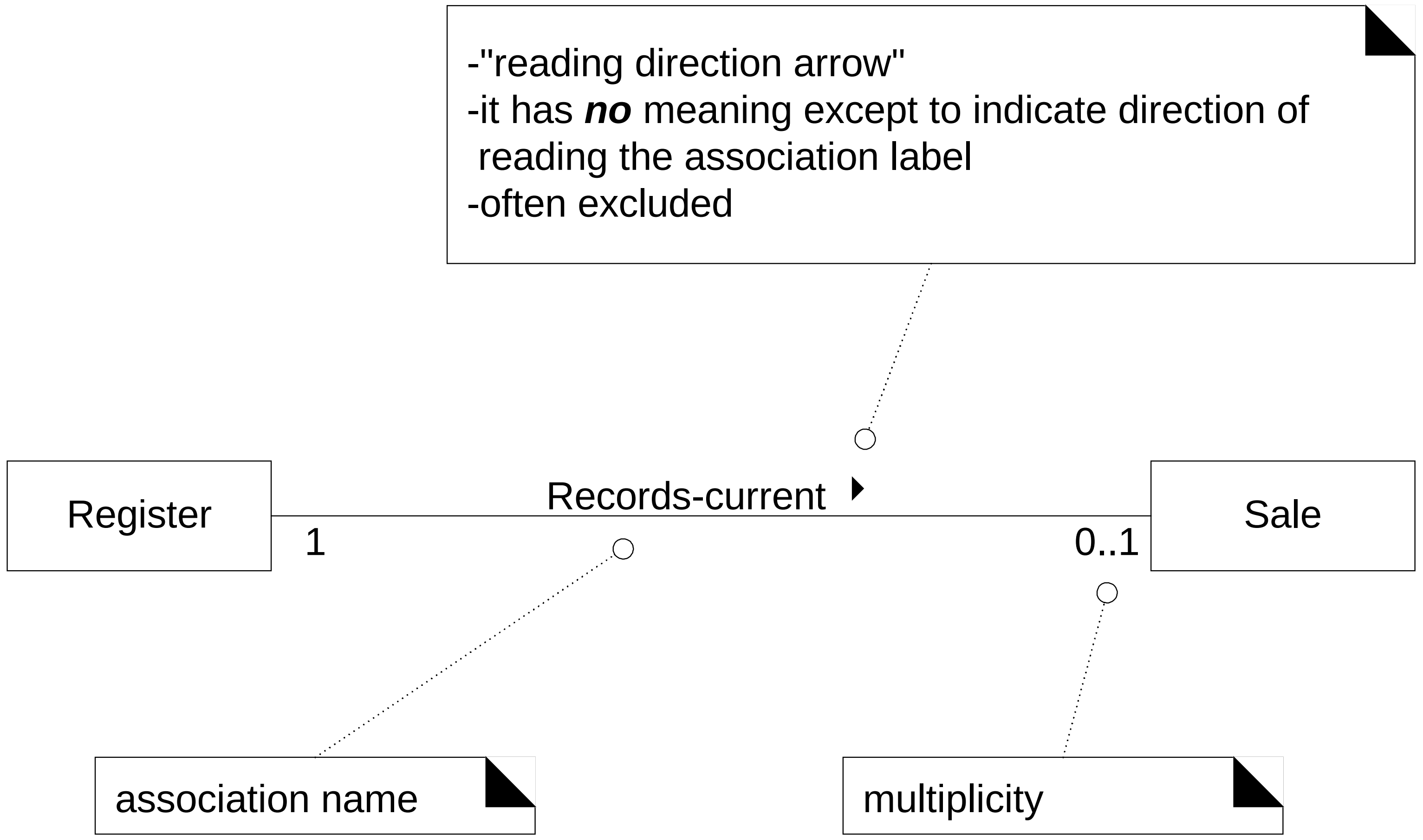
Realize that there may not be a direct association between software classes in the class definition model just because there is an association between conceptual classes in the domain model

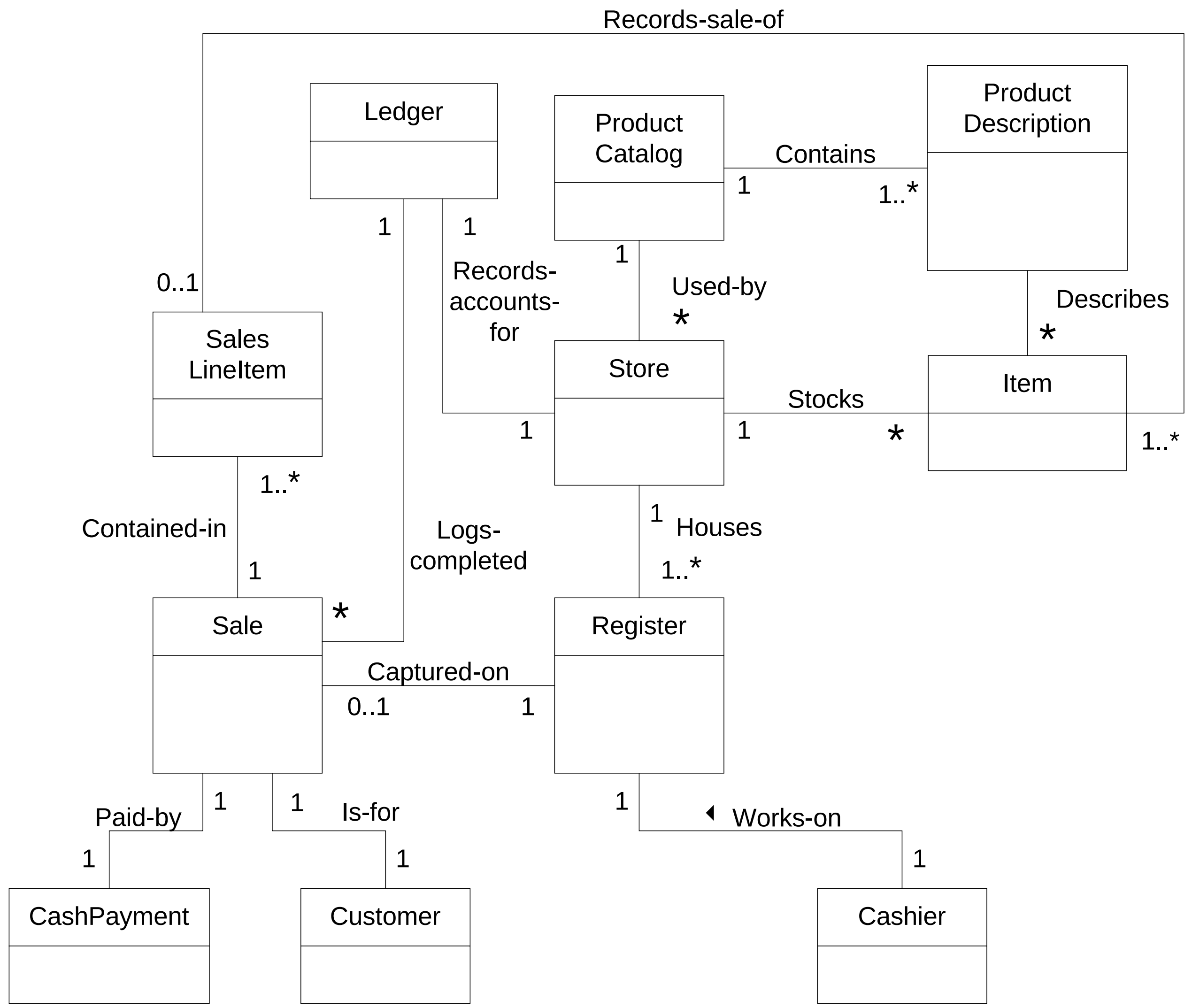
- Associations in the domain model show that the relationship is meaningful in a conceptual way
- But many of these relationships do become paths of navigation in the software

Naming: Use *ClassName – VerbPhrase – ClassName* format

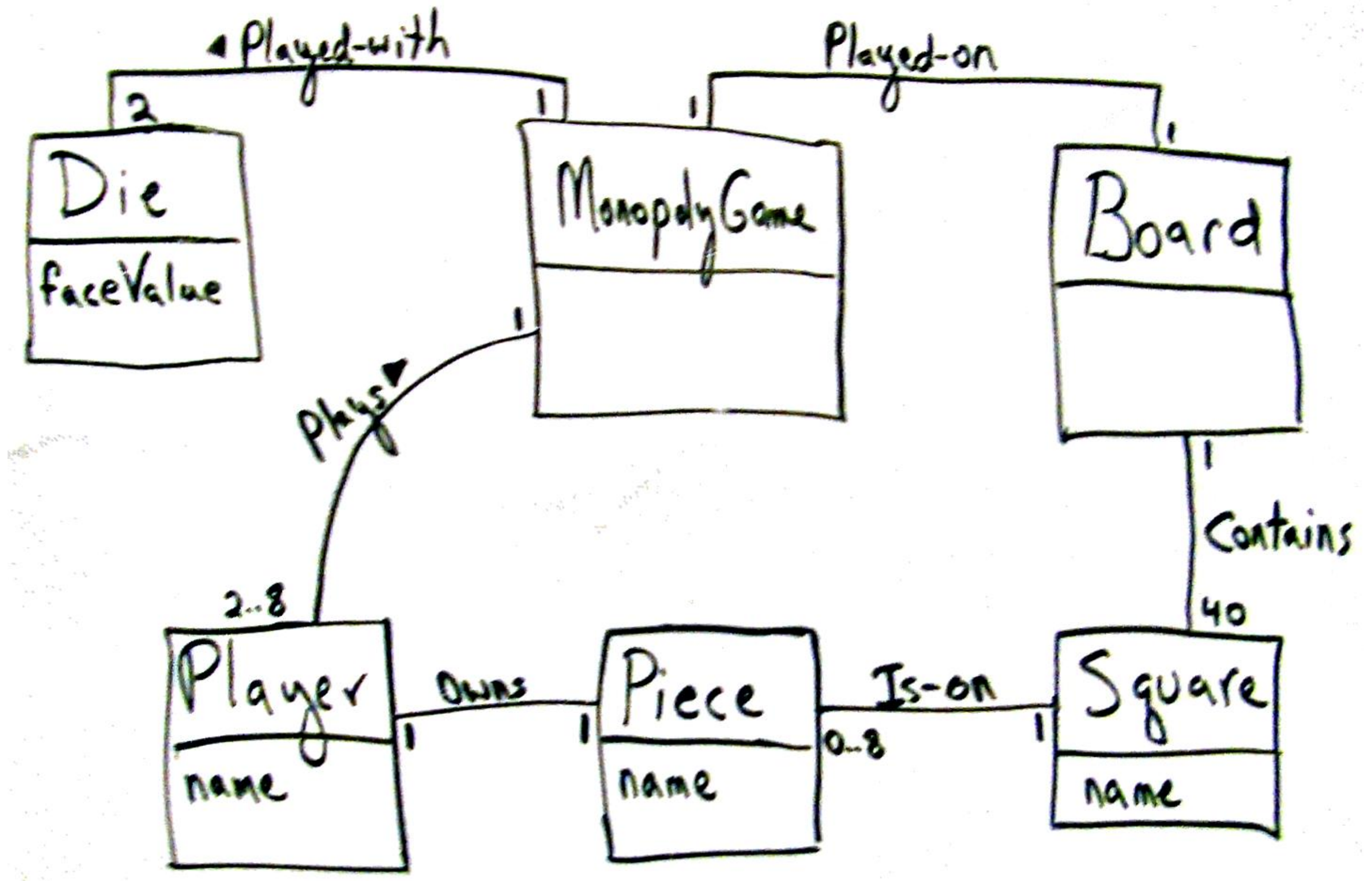
Can add a small arrow help to help explain the diagram to the reader

Associations


-"reading direction arrow"
-it has *no* meaning except to indicate direction of reading the association label
-often excluded
Register
Records-current ▸
Sale
1
0..1
association name
multiplicity



NextGen POS – Domain Model with associations



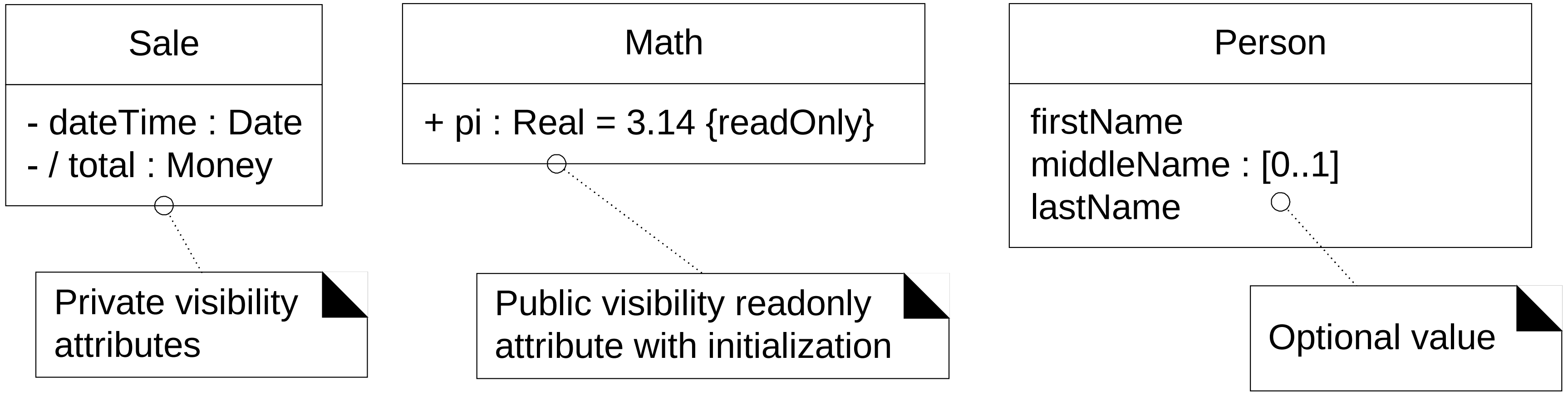
Monopoly Game – Domain Model with associations

Domain Models: Adding Attributes

Useful to add attributes to conceptual classes to satisfy an information requirement in a scenario. Note that in this case the *attribute* is a logical data value of an object

Attributes are added to the bottom of the conceptual class box

Notation: **visibility name : type multiplicity = default value {property-string}**



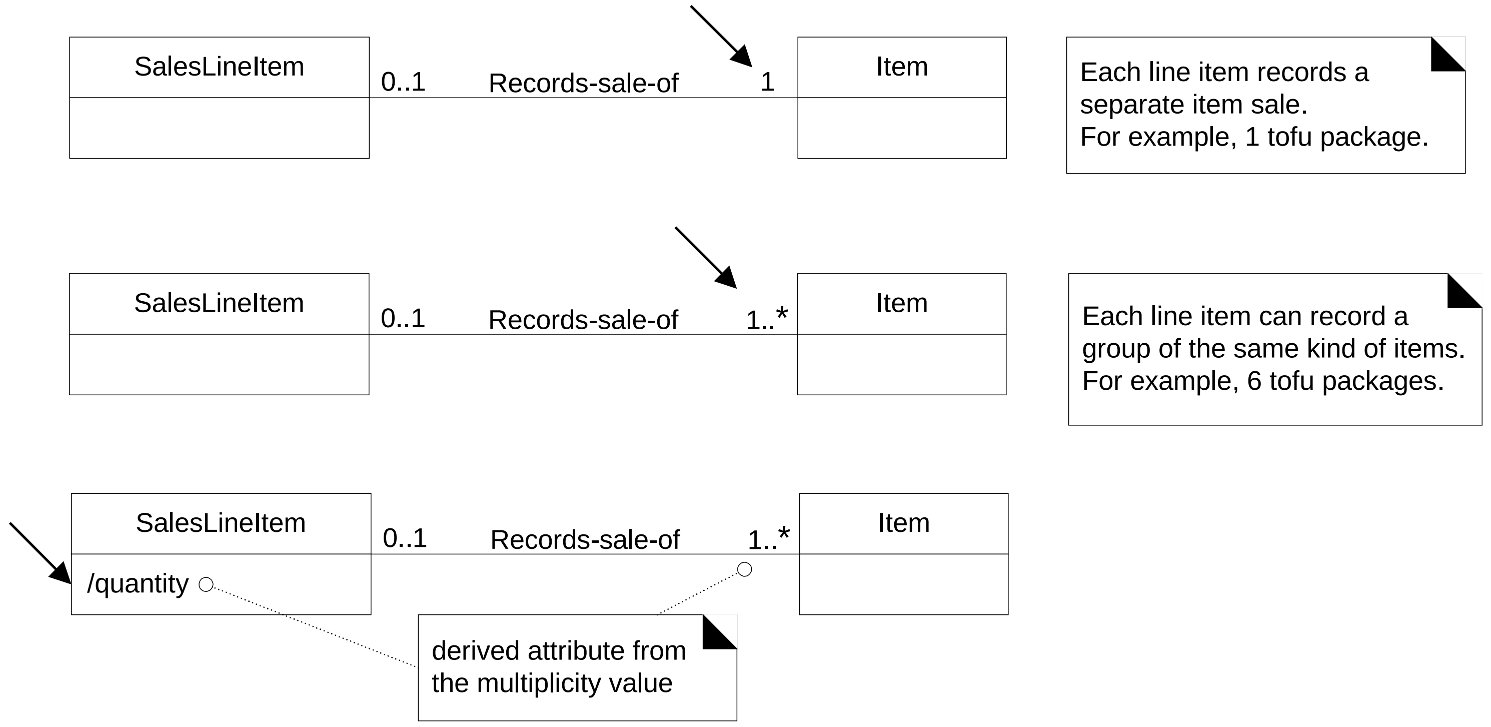
Sale
- dateTime : Date
- / total : Money
Private visibility attributes
Math
+ pi : Real = 3.14 {readOnly}
Public visibility readonly attribute with initialization
Person
firstName
middleName : [0..1]
lastName
Optional value

Domain Models: Adding Attributes

- Usually assume that the attribute is private, unless otherwise noted
- Be careful about placing attribute requirements in the Domain Model
  - The Domain Model is generally used as a tool to understand the system under development, and often any requirements that are captured there may be overlooked
  - Best to capture attribute requirements in a Glossary
  - Can also use UML tools that can integrate a data dictionary
- Note in the previous example we use the symbol “/” to indicate that an attribute is *derived*, i.e. computed.

Derived Attribute: Example

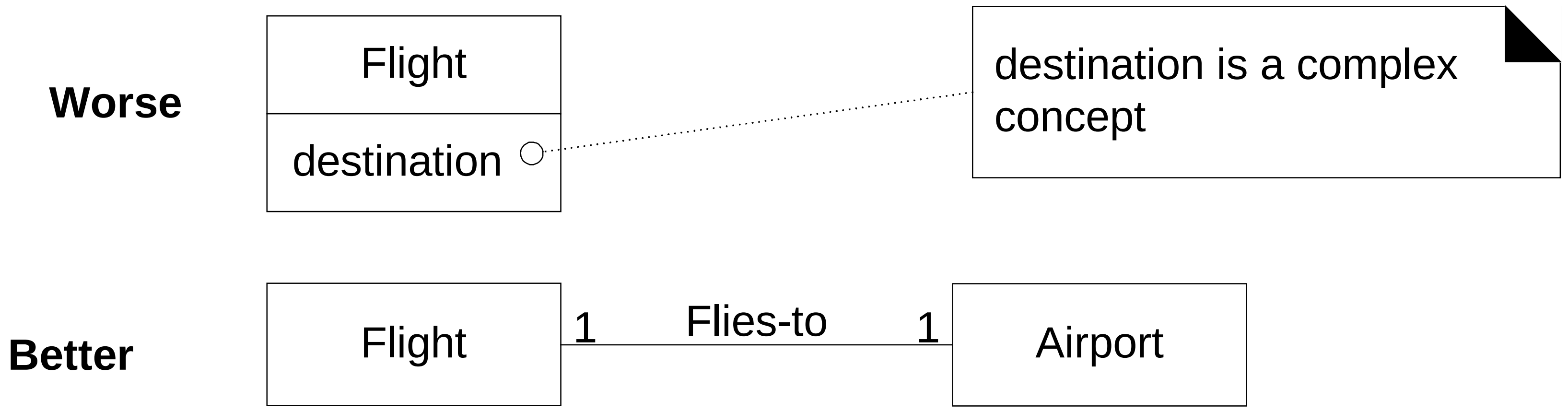
In this case, multiple instances of an item can be added to a SaleLineItem one at a time or as a group. The *quantity* attribute can be computed directly from the multiplicity of the Items:



SalesLineItem
0..1
Records-sale-of
1
Item
Each line item records a separate item sale. For example, 1 tofu package.
SalesLineItem
0..1
Records-sale-of
1..*
Item
Each line item can record a group of the same kind of items. For example, 6 tofu packages.
SalesLineItem
/quantity
0..1
Records-sale-of
1..*
Item
derived attribute from the multiplicity value

Attributes Versus Classes

- Often attributes are primitive data types
  - Boolean, Date, Number, Char, String, Time, …
- Do not make a complex domain concept an attribute – this should be a separate class.
  - Data types are things which can be compared by value; conceptual classes are usually compared by identity
  - Person class versus name string



Data Type Classes

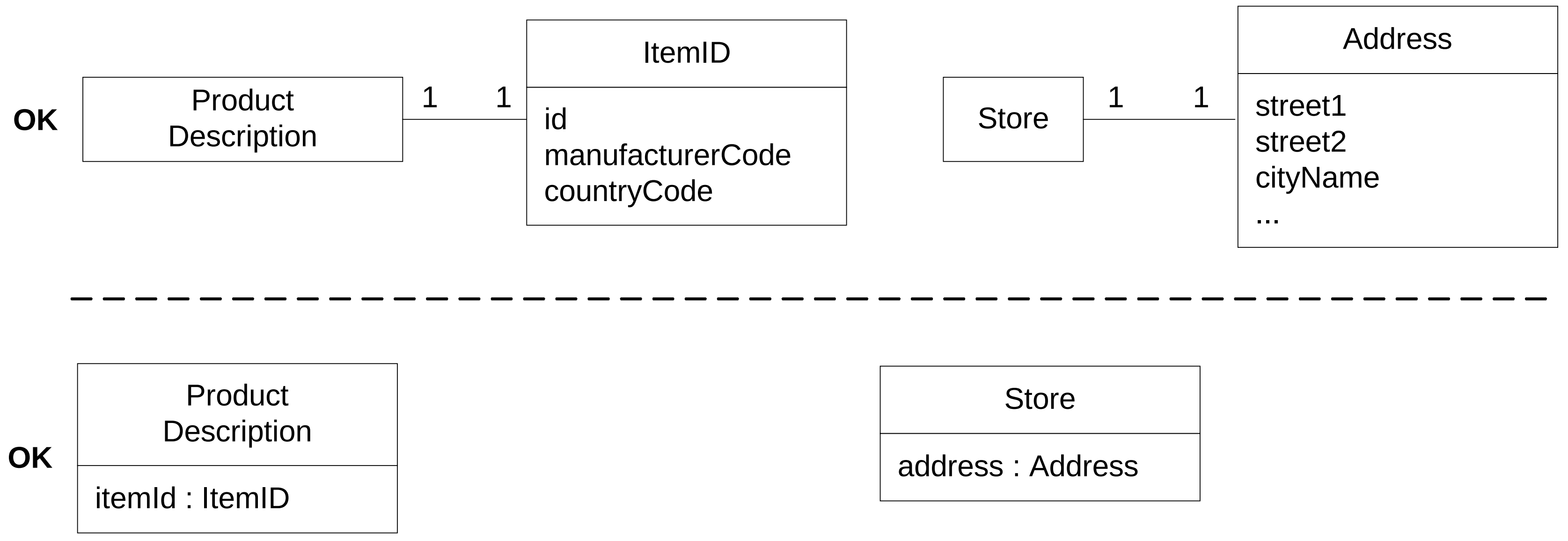
It is also possible to have more complex data types as attributes in the Domain Model, and these are often modeled as classes

For example, in the NextGen POS example, we may have an *itemID* for each item; it is probably contained in the *Item* or *ProductDescription* classes. It could be a number or a string, but it may have more parts too

In general, your analysis of the system will tell if the attributes are simple or need more complexity

- For example, upon examining the detail of the *itemID*, we may discover that it is made up of multiple parts, including a unique UPC, a manufacturer ID, a country code, etc. This would be a good candidate for a separate class

Data Type Class: Example



In the bottom example, *itemID* and *address* would need to be described in the Glossary or someplace else in the Domain Model

Guidelines for Creating Data Type Class

If the data type is composed of multiple sections, like *name, phone number,* etc.

There are operations associated with the data type, like parsing

There are other attributes that are associated with it

- A *promotionalPrice* may have a *startDate* and an *endDate*.

It represents a quantity with a unit, e.g. currency

It is an abstraction with one or more types of the above

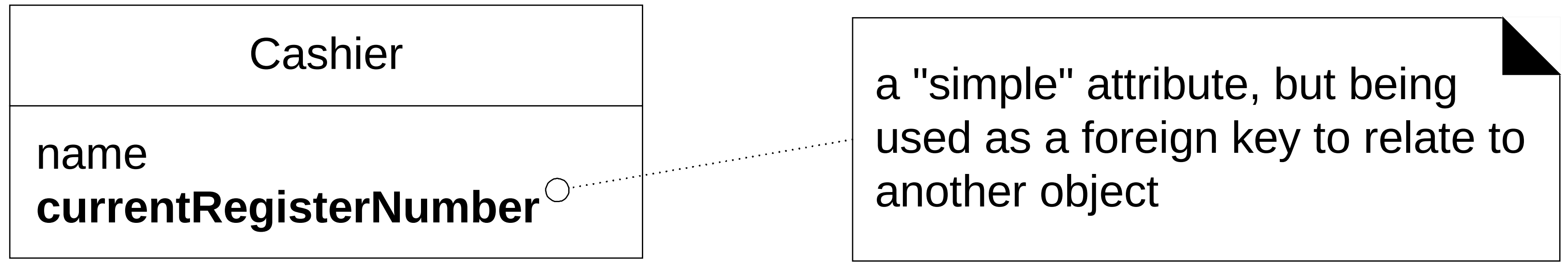
Do not use a “foreign key” to associate two classes – use UML associations, not attributes

- A simple attribute that is used to relate two classes – see next slide for an example

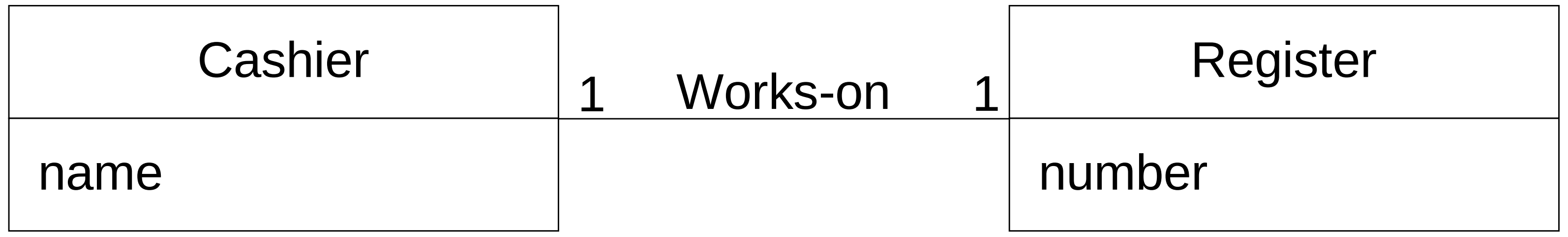
Be careful with quantities that require units – best to do a separate classes. Examples are *Money* and *Weight*

No Foreign Keys

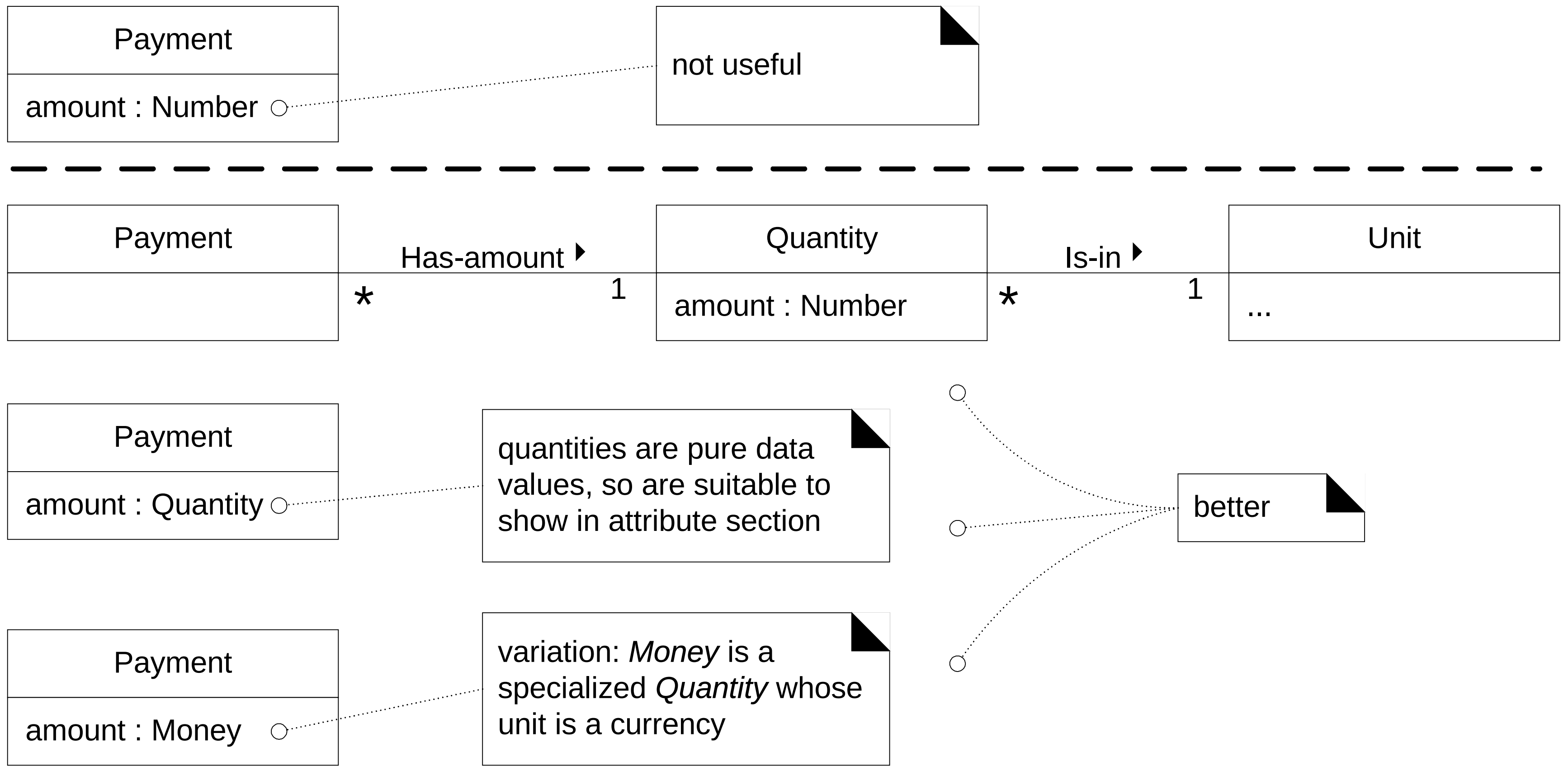
Worse

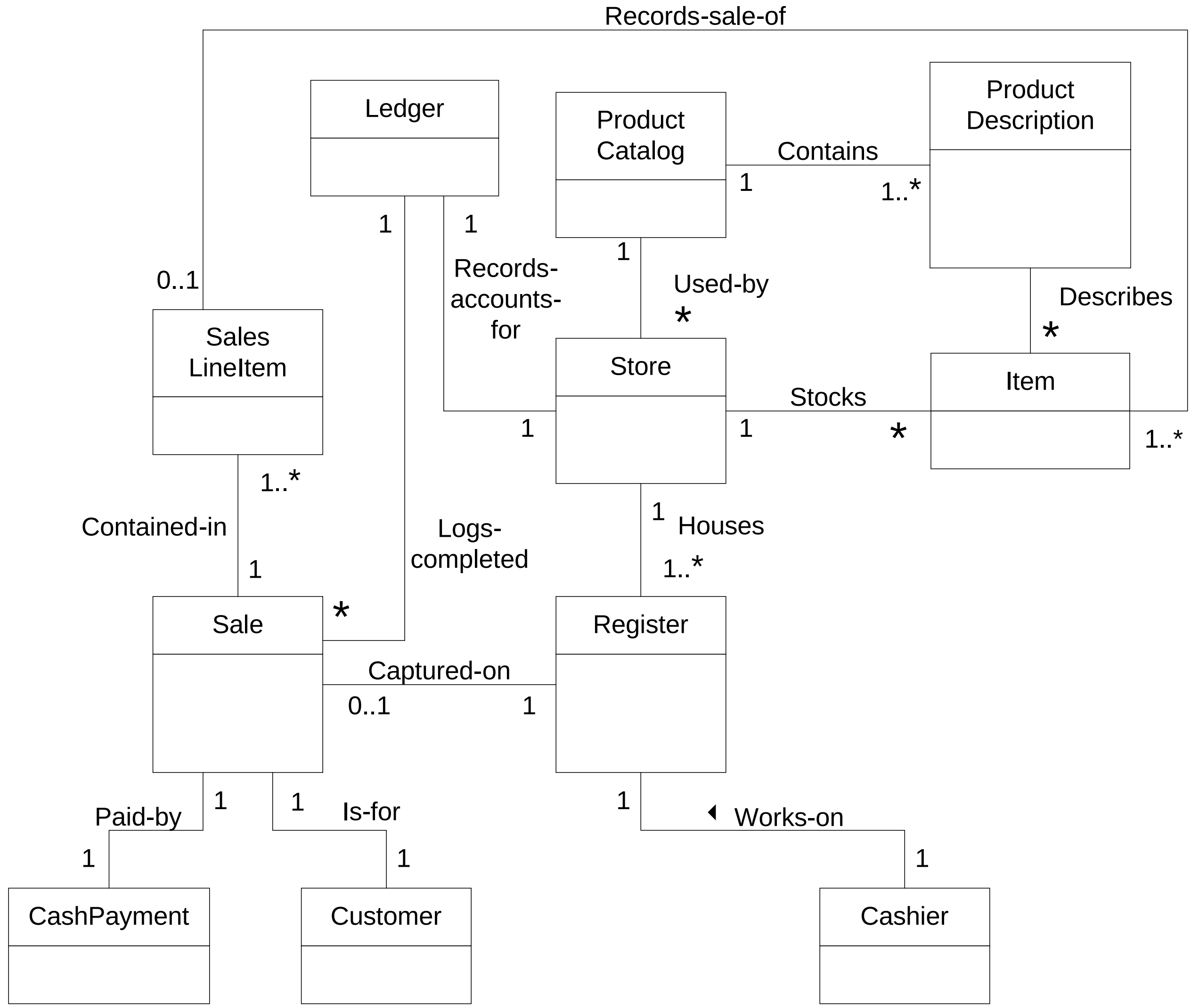

Cashier
name
currentRegisterNumber
a "simple" attribute, but being used as a foreign key to relate to another object

Better



Modeling Quantities


Payment
amount : Number
not useful
Payment
Has-amount
*
1
Quantity
amount : Number
Is-in
*
1
Unit
...
Payment
amount : Quantity
quantities are pure data values, so are suitable to show in attribute section
better
Payment
amount : Money
variation: *Money* is a specialized *Quantity* whose unit is a currency

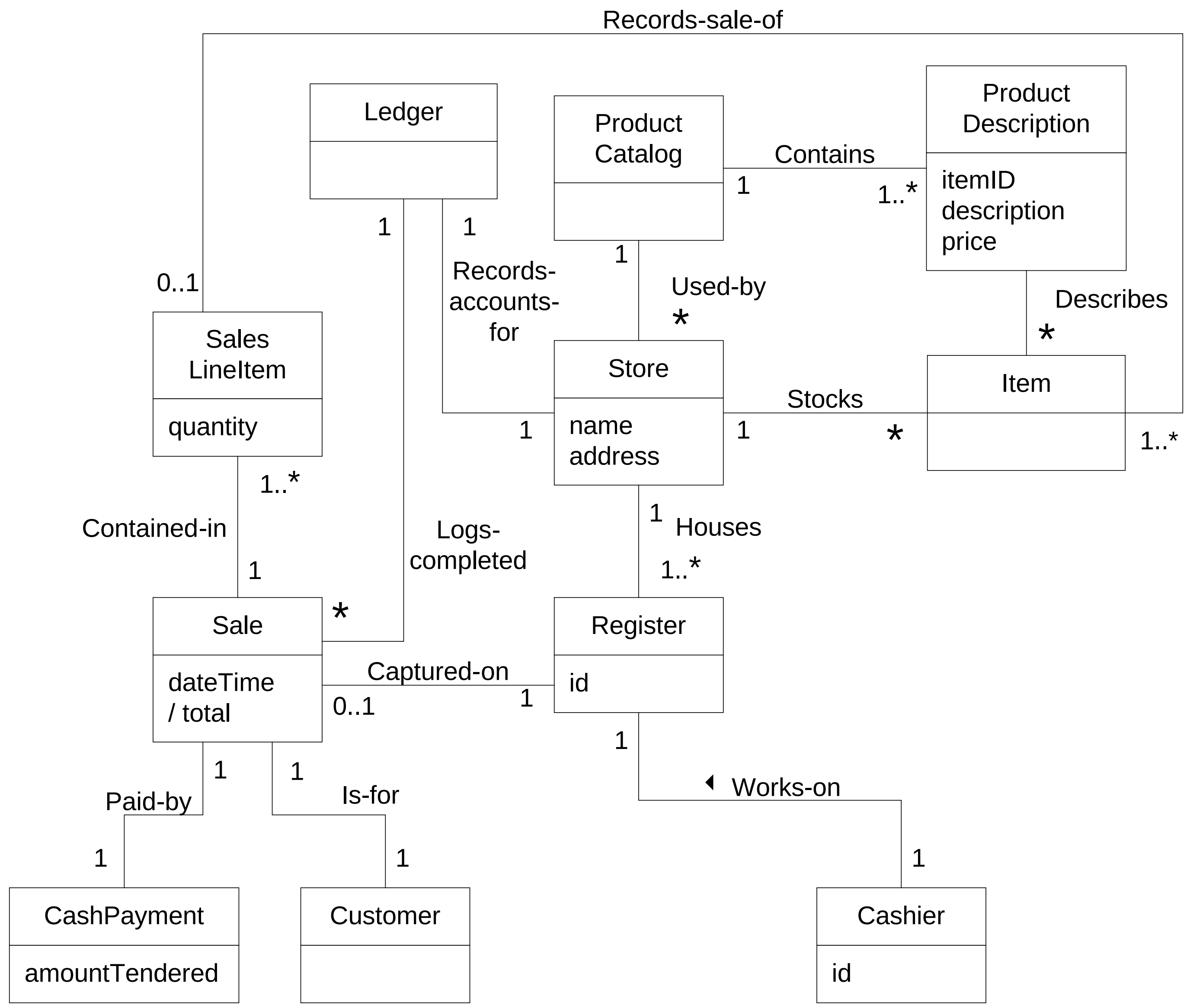


NextGen POS – Domain Model with associations, add attributes?

Attributes: POS Example

After examining the NextGen POS system, we may conclude the following attributes are needed (we are only assuming cash payments in this iteration):

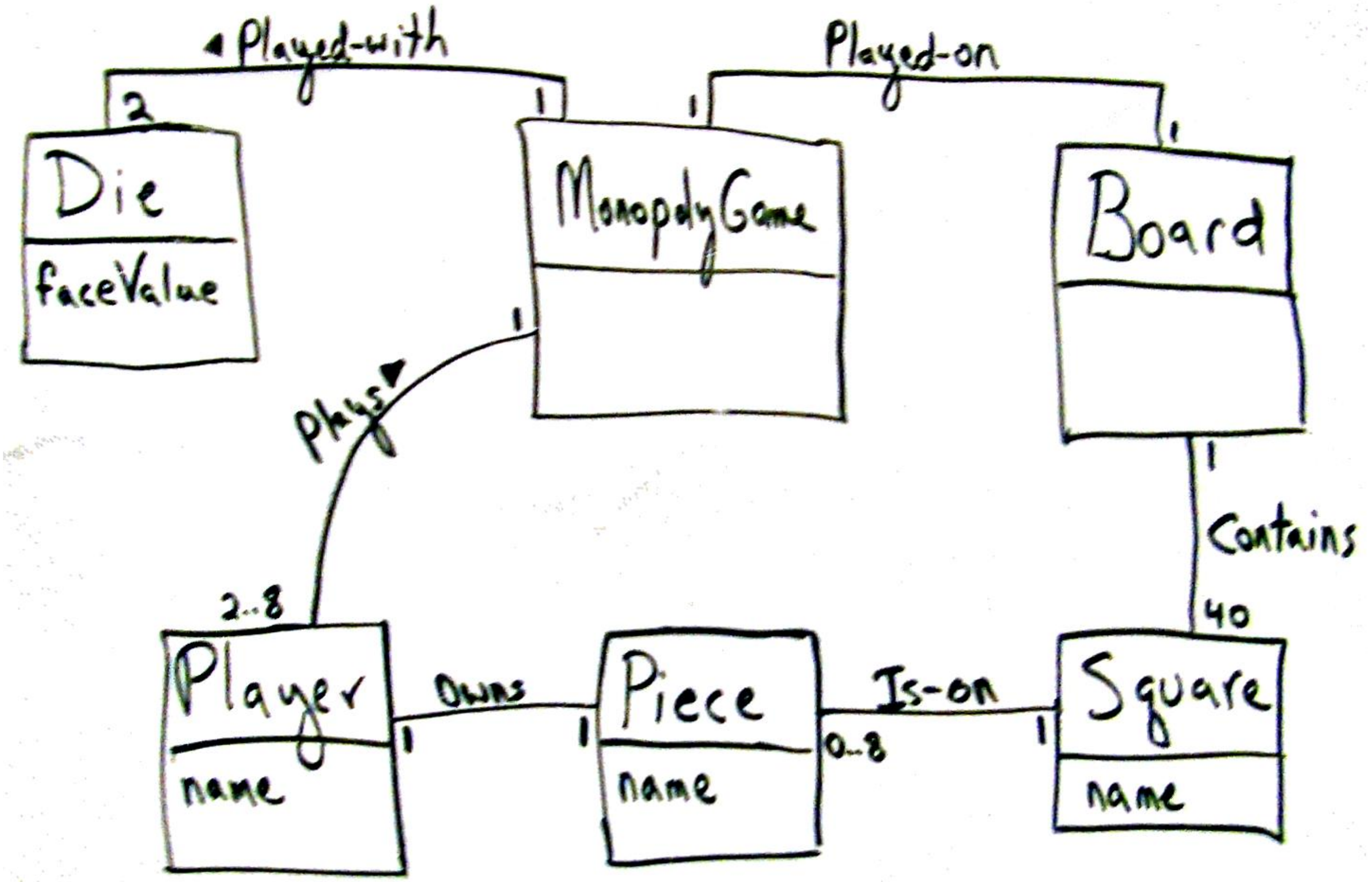
- *CashPayment* – Will contain an *amountTendered*, which is usually a currency amount, must be captured
- *ProductDescription* – This class will need several pieces of information to accurately describe a an item: *description, itemID,* and *price* for starters.
- *Sale* – this class will need a *dateTime* attribute. Also will contain a derived *total* attribute.
- *SalesLineItem* – this class will need to include a quantity attribute
- *Store* – this class will need to contain attributes for *name, address*


Records-sale-of
Ledger
Product Catalog
Product Description
itemID
description
price
Contains
1
1..*
1
1
Records-accounts-for
1
Used-by
*
Describes
*
0..1
Sales LineItem
quantity
Store
name
address
Stocks
1
*
Item
1..*
1
1..*
Contained-in
1
Logs-completed
1
Houses
1..*
Sale
*
dateTime
/ total
Register
id
Captured-on
0..1
1
1
1
1
Paid-by
Is-for
Works-on
1
1
1
CashPayment
amountTendered
Customer
Cashier
id

Attributes: Monopoly Example

After examining the Monopoly system, we may conclude the following attributes are needed (remember, this is just the first iteration):

- *Die* – This class will need a *faceValue* attribute, so we can print the players’ dice rolls to the log
- *Square* - this class will need a *name* attribute, so we can print the players’ moves to the log
- *Player* – this class needs a *name* for logging
- *Piece* – this class needs a *name*


◄ Played-with
2
1
Die
faceValue
MonopolyGame
1
Played-on
1
Board
1
Plays ►
2..8
Player
name
Owns
1
1
Piece
name
Is-on
0..8
1
Square
name
Contains
1
40

Domain Modeling in UP

Models are tools of communication and understanding, and by nature they are inaccurate representations of the real world

- There is no “correct” Domain Model, but some models are more useful than others

The Domain Model will evolve over several iterations in the UP

- The model is usually limited to prior and current scenarios under construction as use cases are developed

Rarely is the Domain Model started during Inception – it is usually started during the early stages of Elaboration, and evolves over the first few iterations as the requirements and use cases are revealed

UP also has a *Business Object Model* artifact, which is an enterprise model used to describe the entire business

- The BOM artifact consists of several items, including diagrams (class, activity, sequence) that show how the enterprise should run
- We will not study this artifact

Domain Model Refinement

Suppose in our next iteration of the NextGen POS system, as the result of our analysis we add two new payment options: By credit card and by check. The *Process Sale* use case has been extended:

Use Case UC1: Process Sale

…
Extensions
7b Paying by Credit:
1. Customer enters their credit account information
2. System sends payment authorization request to an external Payment Authorization Service, requests payment approval
3. System receives payment approval

…

7c Paying by Check
1. Customer writes check, and give it along with their drivers license to the Cashier
2. Cashier writes the drivers license number on the check, enters it, and requests check payment authorization

…

Domain Model Refinement: Generalization

Note that we could go through this use case and identify new concepts based upon the nouns we see

- CreditAuthorizationService, CheckAuthorizationService, etc.

Also note that the concepts of CashPayment, CreditPayment, and CheckPayment are all very similar – they just represent different ways of paying

In situations like this, it is often useful to organize the various related concepts into a *generalization-specialization class hierarchy* (or *class hierarchy*).

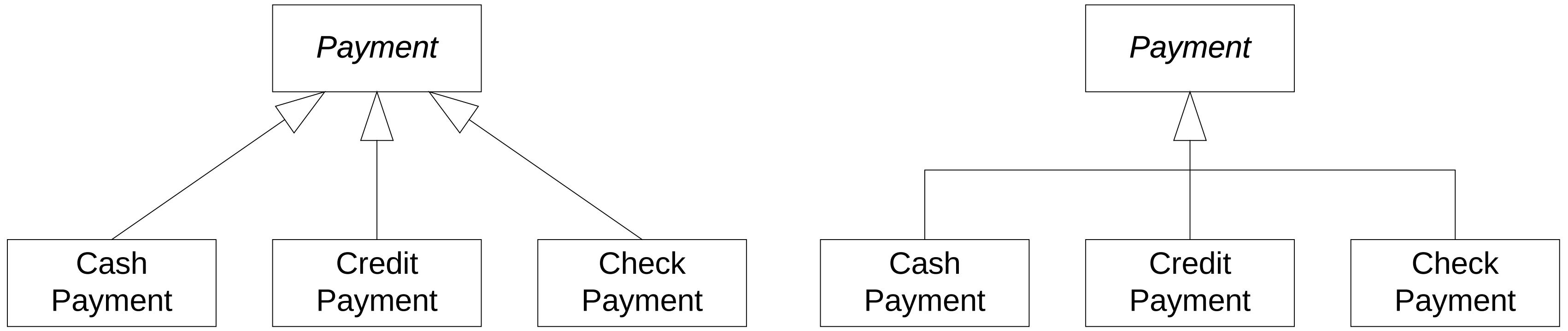
In this example, it would make sense to create a *Payment* superclass, and consider the cash, credit, and check payments to be specialized subclasses

Remember – we are in the Domain Model, so these are conceptual classes, not software classes

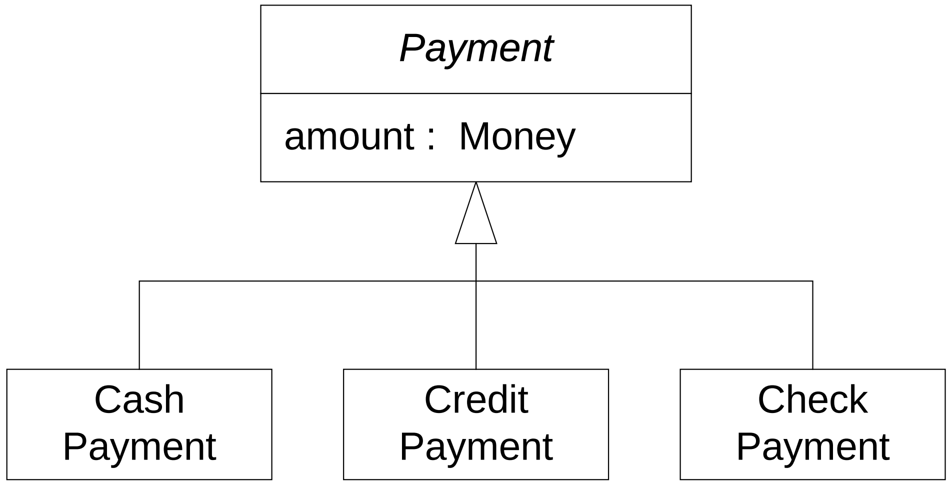
Domain Model Refinement: Generalization

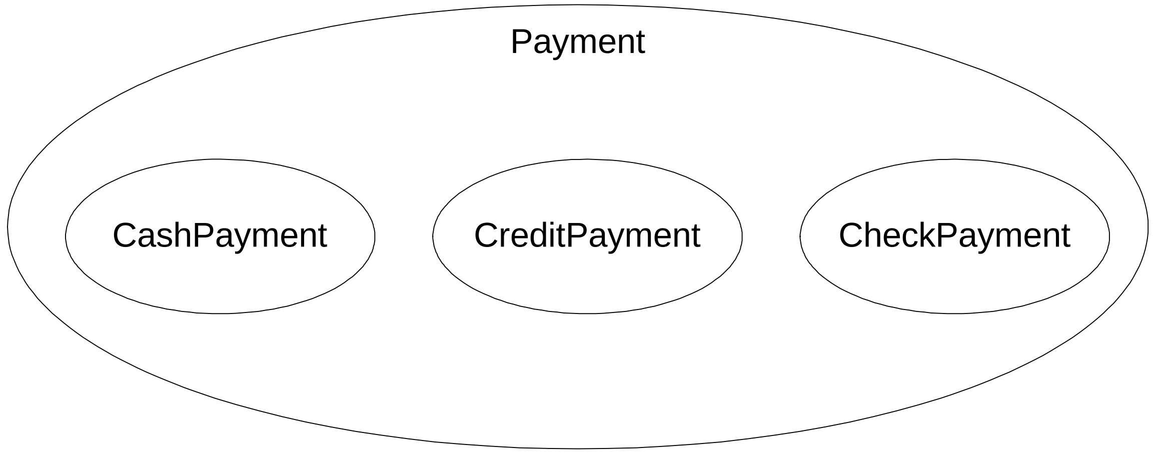
Even though these are not software classes, this type of modeling can lead to better software design later (inheritance, etc.)

UML notation uses an open arrow to denote subclasses of a conceptual class



Super- and Sub-Classes

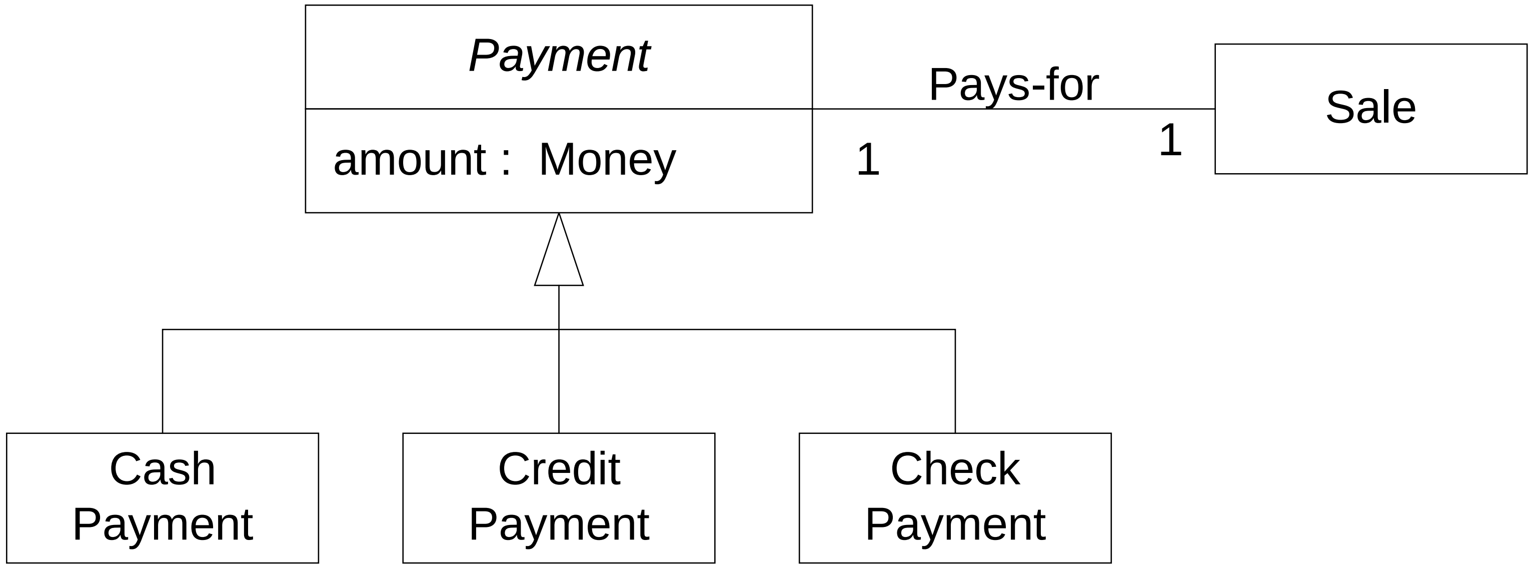




Super and Sub Classes

All statements about the super-class apply to any sub-classes; in other words, 100% of the super-class definition applies to the sub-class

- The sub-class must conform to the attributes and associations of the super-class
- A Credit Payment pays-for a Sale. A Check Payment has an amount



Refining Domain Models: Super and Sub Classes

Can think of the "sub-class as being *a kind of* super-class"

- A Credit Payment is a kind of Payment
- Often shortened to: A Credit Payment *is a* Payment

We can identify sub-classes by these two rules

- The 100% - all the super-class definition applies to the sub-class
- The "is a" rule – the sub-class *is a* kind of super-class

Any sub-class of a super-class must obey these rules

Sub-Classes: When to Define

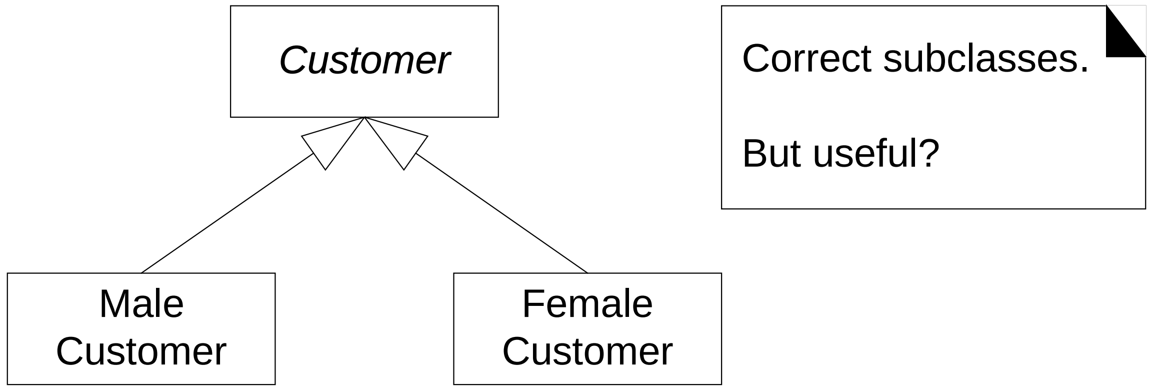
Occasionally a super-class is created to capture similar sub-classes, but usually the process works the other way: A more general super-class has been defined, and as the iterations proceed, sub-classes are created to handle more detailed scenarios

Guidelines: Create a sub-class when …

- The class under consideration "is a" kind of an existing super-class
- The class under consideration has additional attributes of interest
- The class under consideration has additional associations of interest
- The class under consideration is operated on, handled, reacted to, or manipulated differently than the super-class (or other subclasses)
- The class under consideration represents an actor that behaves differently than the super-class or other sub-classes

Super- and Sub-Classes: When to Create

Bad example:



When would this make sense?

- Market research model, where there are behaviors of male and female shoppers that are different
- Medical research, since men and women are different biologically

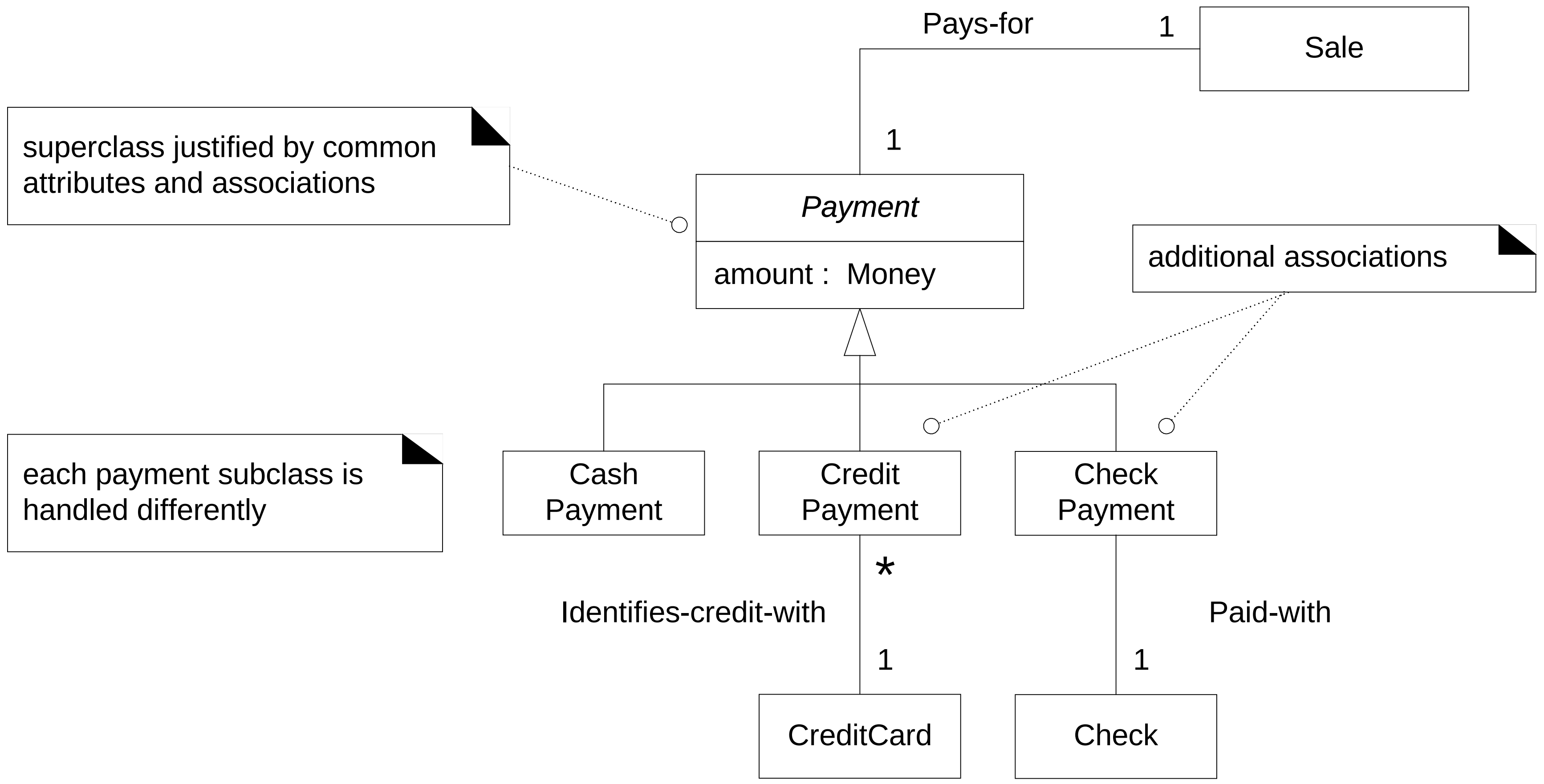
Super-Classes: When to Define

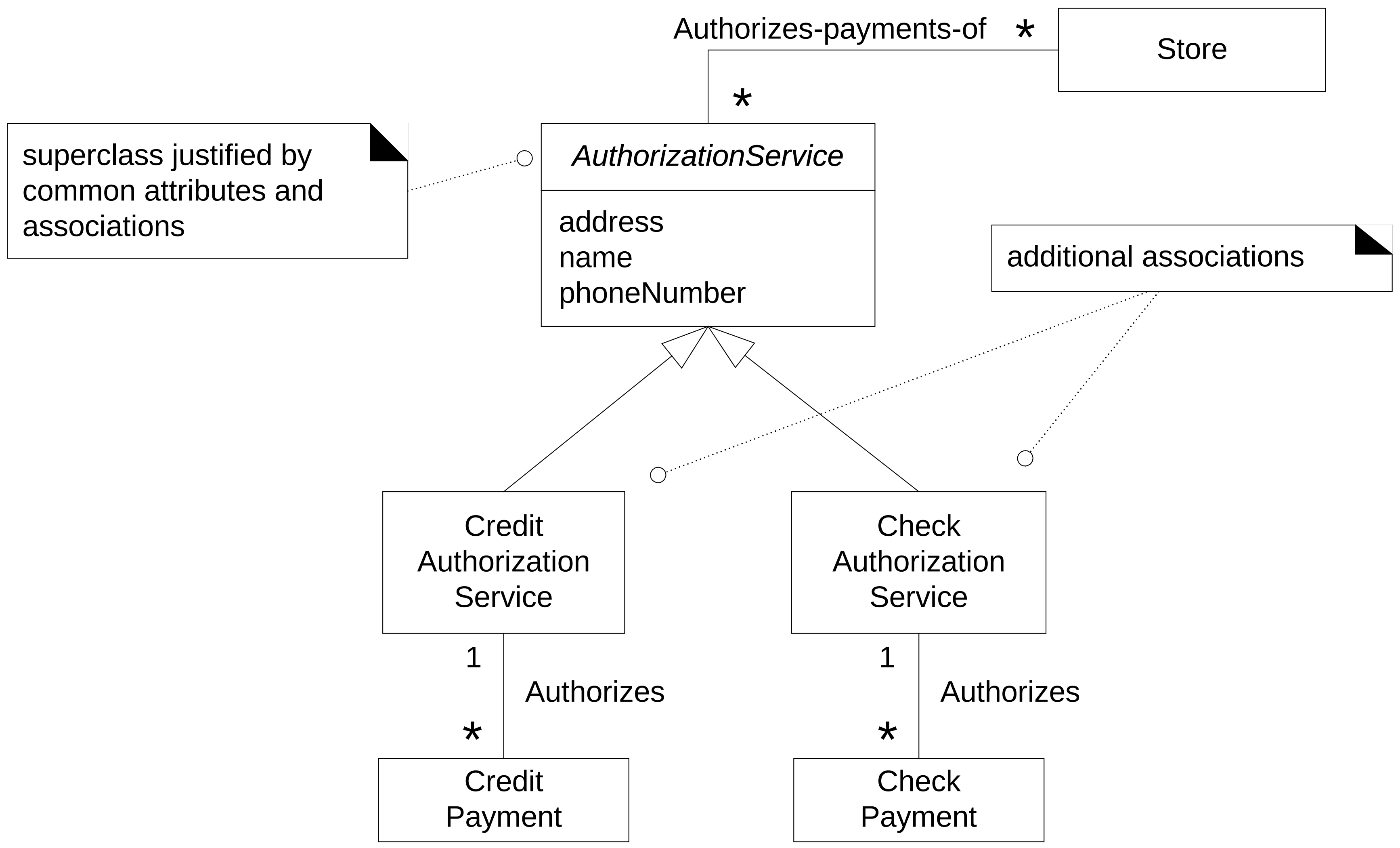
This may occur when common traits are identified among various conceptual classes in the Domain Model

Create a super-class when:

- The potential sub-classes appear to be variations of the same concept
- All the potential sub-classes will conform to the “is a” rule for the new super-class
- A set of common attributes that belong to all the potential sub-classes is identified and can be factored out
- A set of common associations that all potential sub-classes have has been identified and can be factored out

Example: Payment Sub-Classes




Authorizes-payments-of
*
Store
*
superclass justified by common attributes and associations
AuthorizationService
address
name
phoneNumber
additional associations
Credit Authorization Service
Check Authorization Service
1
Authorizes
*
1
Authorizes
*
Credit Payment
Check Payment

Domain Models: Levels of Granularity

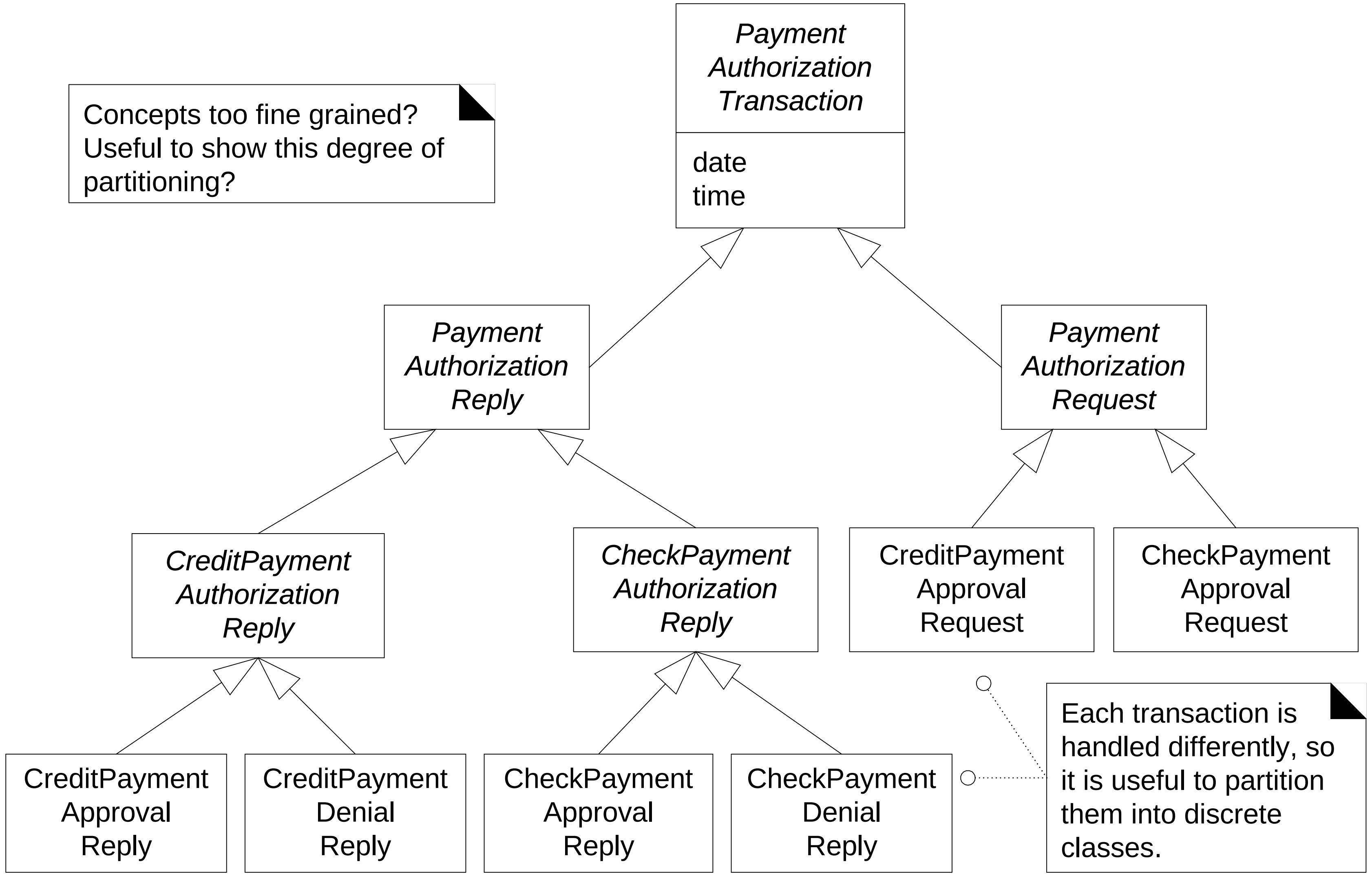
As the Domain Model evolves, the question will arise: What level of detail to go to? How many sub-classes?

Depends on the system being modeled – remember, the Domain Model is a tool to help understand the system

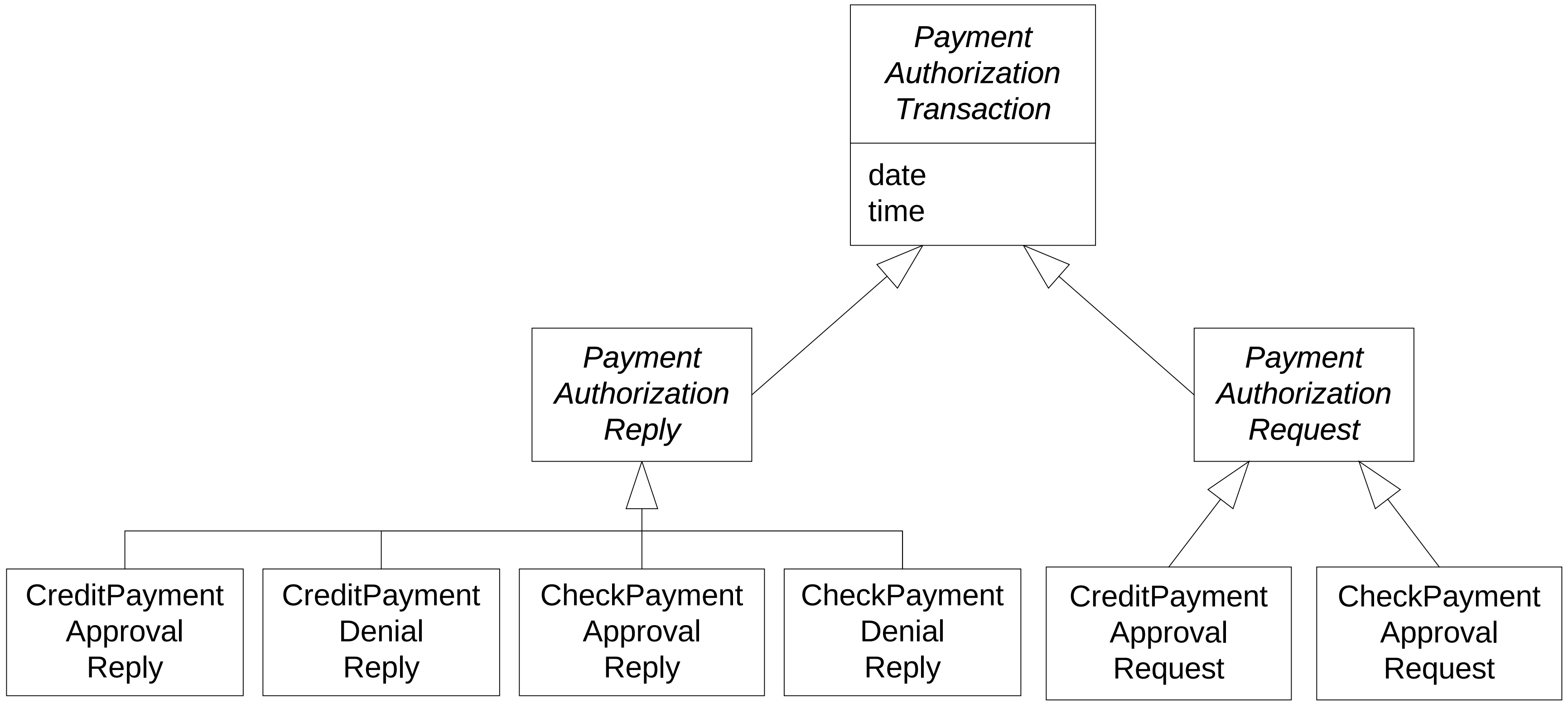
Note that system transactions (request – reply) are usually modeled, because other activities and processes depend upon them

But it is not always necessary to define sub-classes for every transaction

General rule: don’t add complexity unless needed!



Concepts too fine grained? Useful to show this degree of partitioning?
Payment Authorization Transaction
date
time
Payment Authorization Reply
Payment Authorization Request
CreditPayment Authorization Reply
CheckPayment Authorization Reply
CreditPayment Approval Request
CheckPayment Approval Request
CreditPayment Approval Reply
CreditPayment Denial Reply
CheckPayment Approval Reply
CheckPayment Denial Reply
Each transaction is handled differently, so it is useful to partition them into discrete classes.


Payment Authorization Transaction
date
time
Payment Authorization Reply
Payment Authorization Request
CreditPayment Approval Reply
CreditPayment Denial Reply
CheckPayment Approval Reply
CheckPayment Denial Reply
CreditPayment Approval Request
CheckPayment Approval Request

Abstract Conceptual Classes

Recall that when we were developing use cases, we saw that it may be useful to collect a series of steps that occurs in several use cases and define an *abstract* use case that the other use cases could refer to

We can do something similar for conceptual classes: If every member of a class *C* is a sub-class of *C*, then C is called an *abstract class*. But if there is an instance of *C* that is not a member of a sub-class, then *C* is not an abstract class.

- The *Payment* conceptual class we have been discussing is an abstract class, because all payments fall into one of the three sub-classes

Usually, when a super-class is created from a set of sub-classes, the super-class is abstract

Not all super-classes are abstract classes

Denote with *italics* in Domain Model

Modeling State Changes

Occasionally it is necessary to capture the state of an transaction, for example, in the Domain Model.

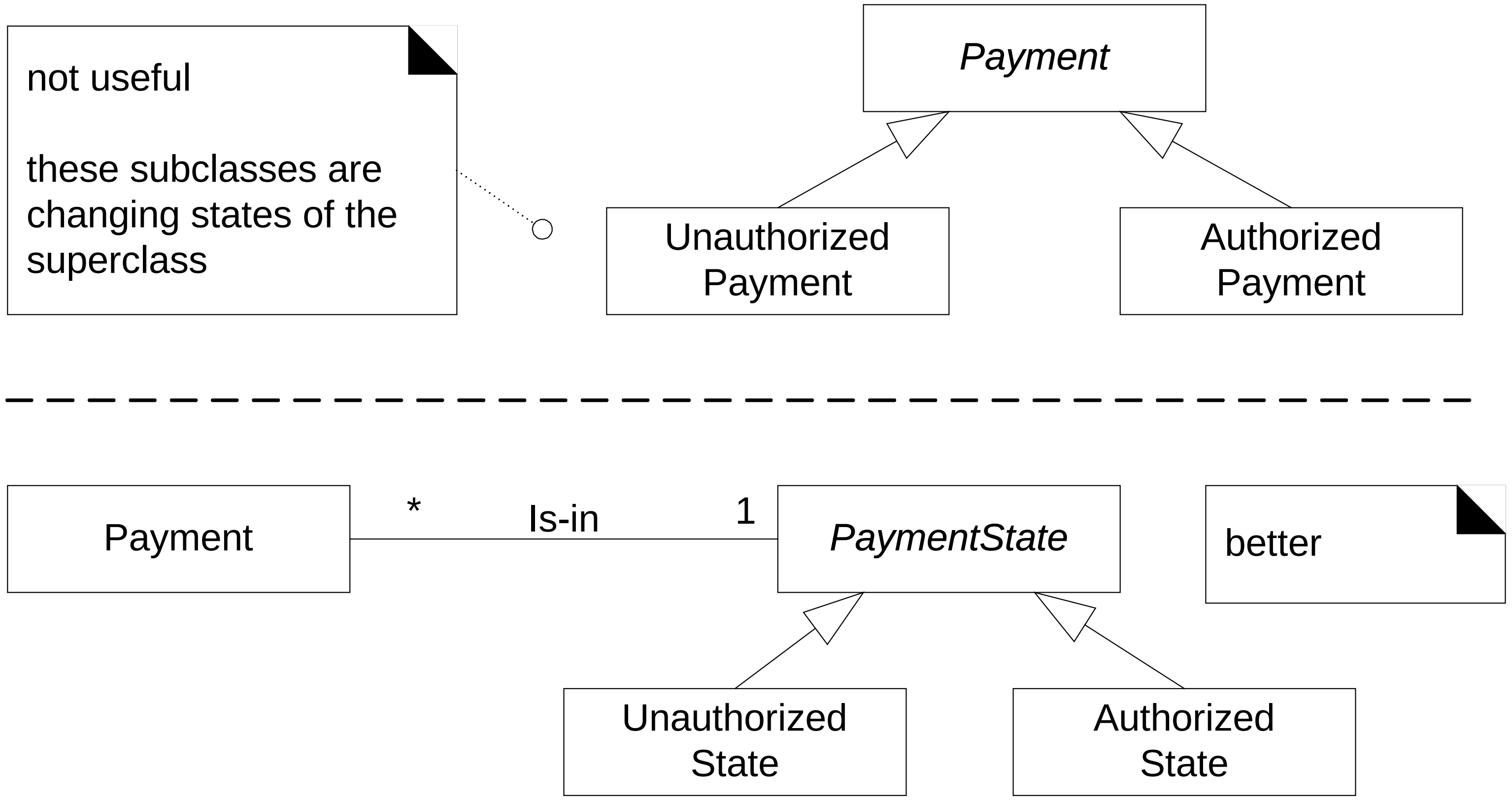
This should not be done with sub-classes, because the super-class can transition

Best to use a separate conceptual class for this, and provide state subclasses of the state class

- The association is usually “is-in” for these state transition classes

State transitions can also be captured in a state diagram

Example: Modeling State Changes



Association Classes

Often, the association between conceptual classes contains information that needs to be captured in the model, but does not belong in either class as an attribute

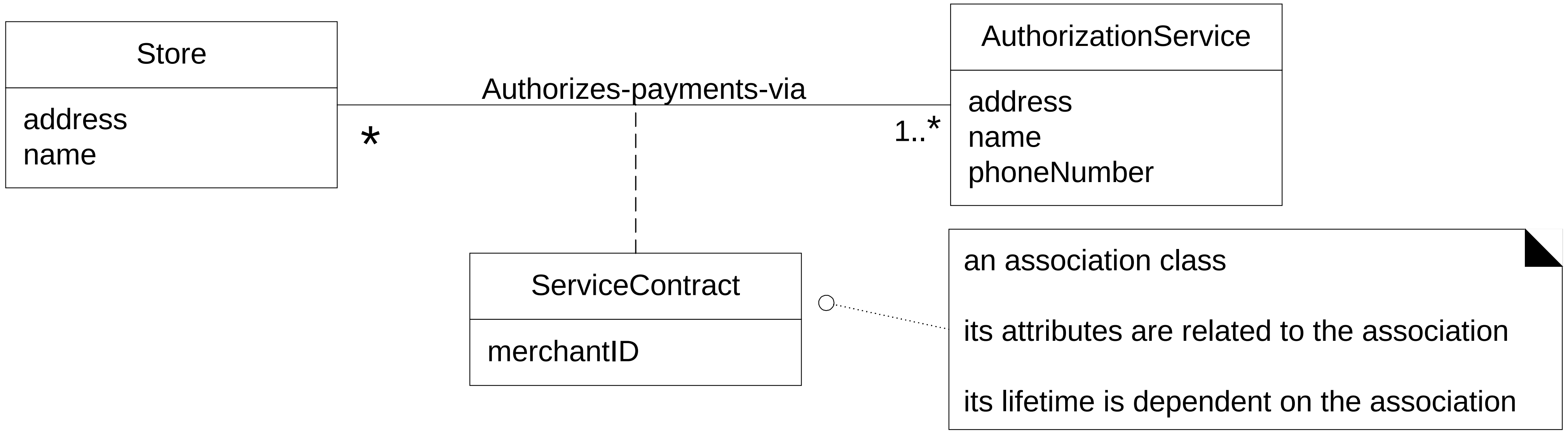
- A merchantID may be an attribute with the payment authorization service, but this does not belong in either the Store or AuthorizationService classes
- A salary may be an attribute of employment, but it does not belong as an attribute of the person or employer classes

General rule: If class *C* can simultaneously have many values of attribute *A*, then *A* should not be placed in *C*.

Could create a new conceptual class and associate it with the existing classes, but this can add complexity to the model

Better way: Create a special class that represents the attributes of the association

Example: An Association Class



Association Classes – When to Add

If the attribute(s) appear to be related to the association and not a particular conceptual class ….

The lifetime of the attributes is dependent upon the association …

There is a many-to-many association between two conceptual classes, and there are many attributes (common clue)

Aggregation and Composition

These are software class concepts that will be important later

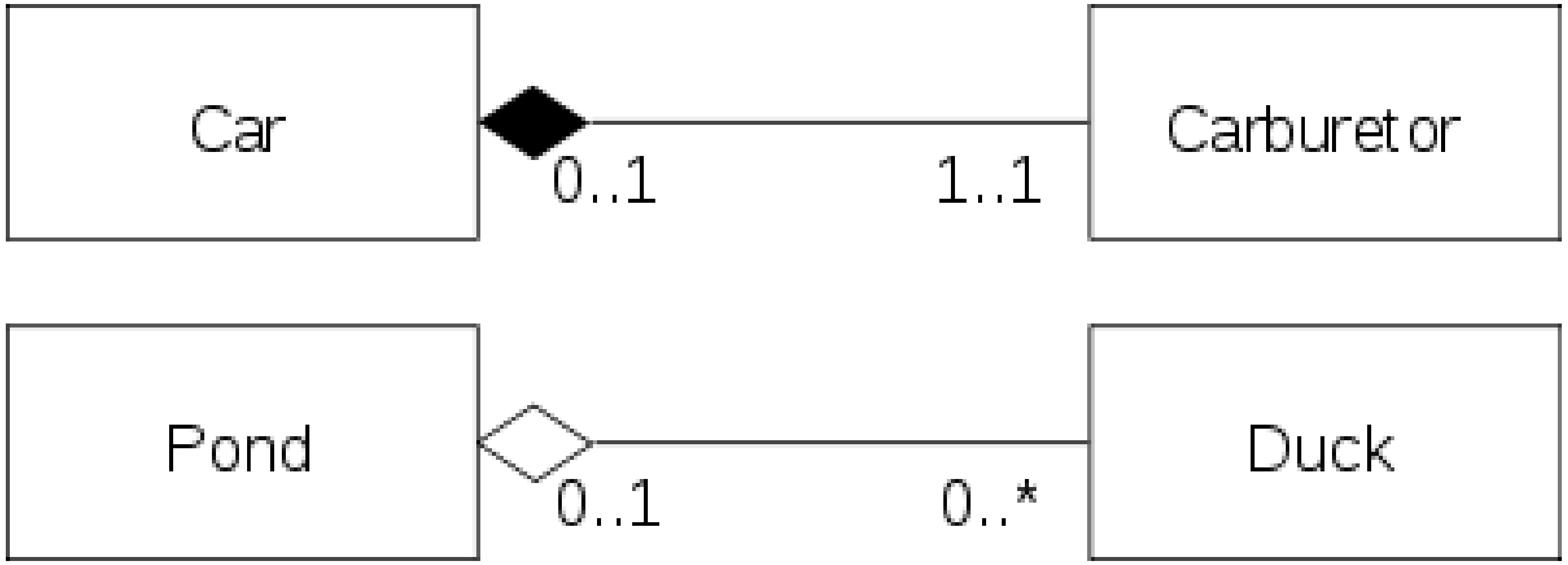
Aggregation implies a container or collection of classes

- In this case, if the container class is destroyed, the individual parts are not
- Denoted in UML as an open diamond

Composition also implies a collection of classes, but with a stronger life dependency

- If the container class is destroyed, the individual component instances are also destroyed
- Denoted by a filled in diamond in UML

Examples: Aggregation and Composition



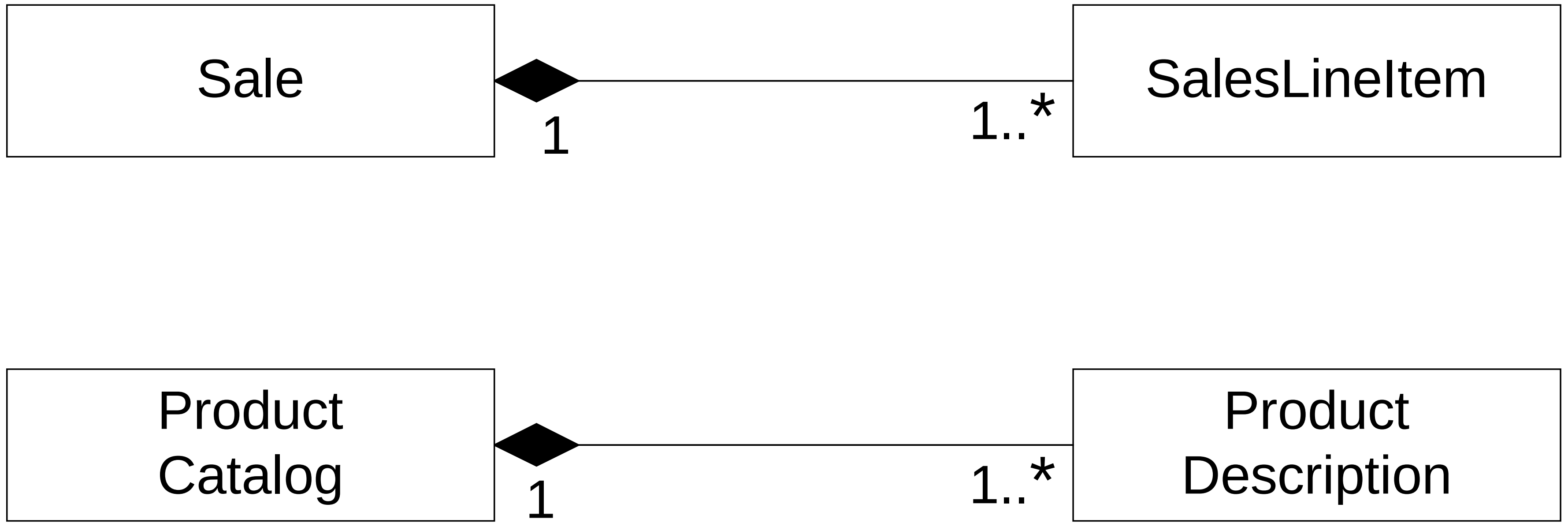
Aggregation and Composition

Usually not critical for domain models, but may be used to …

Clarify constraints in the Domain Model (e.g. existence of a class depends on another class)

Help model situations when create/delete operations apply to many sub-parts

Examples: Composition in NextGen POS

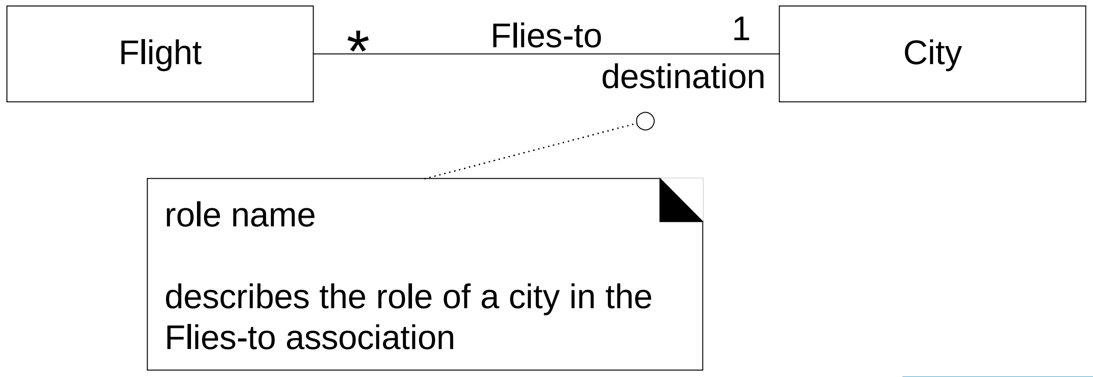


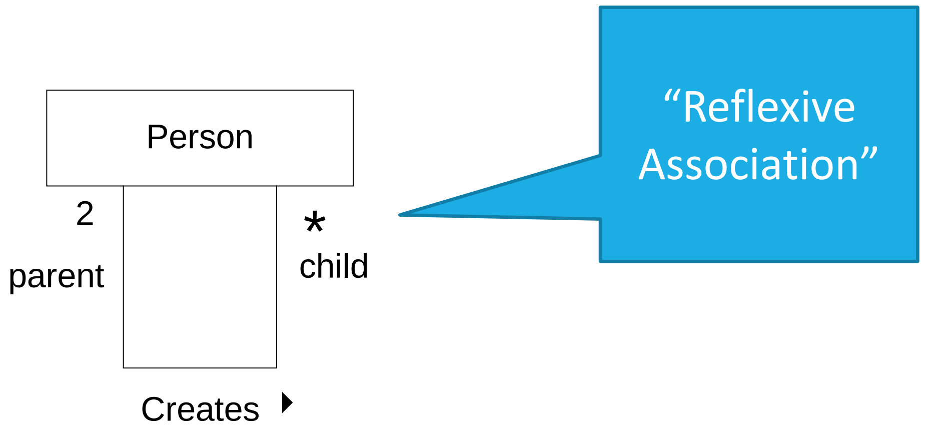
Association Role Names

Occasionally a *role name* is added to an association; this name describes the role the object plays in the association

Not required, often included if there role is not clear

Should model the role as a separate class if there are unique attributes, associations, etc. related to the role

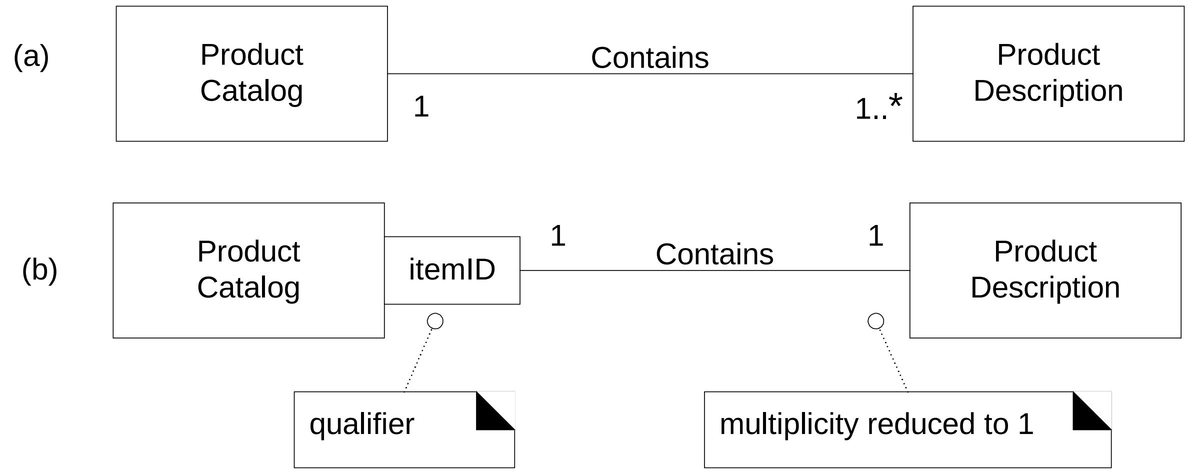




Qualified Associations

A qualifier may be used in an association; it distinguishes a set of objects at the far end of the association based upon the qualifier value

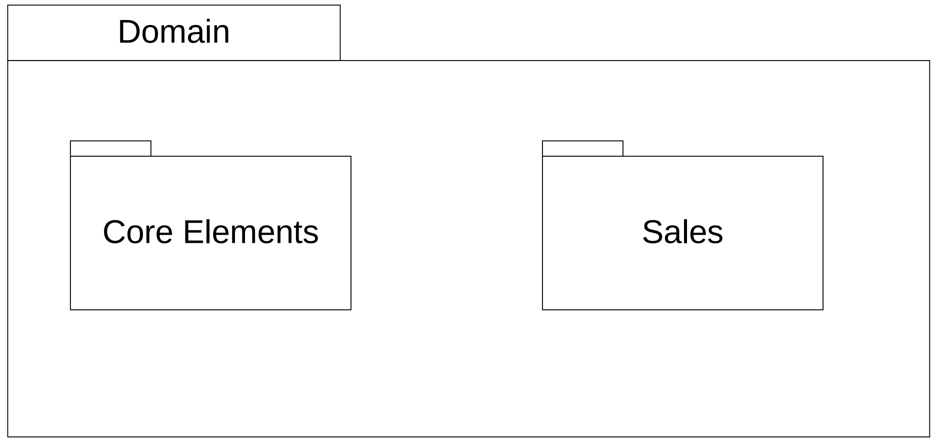
In the example below, we can identify the *ProductDescriptions* by the *itemID*, so we denote this in the UML diagram and change the multiplicity. Be careful not to design!



Packages

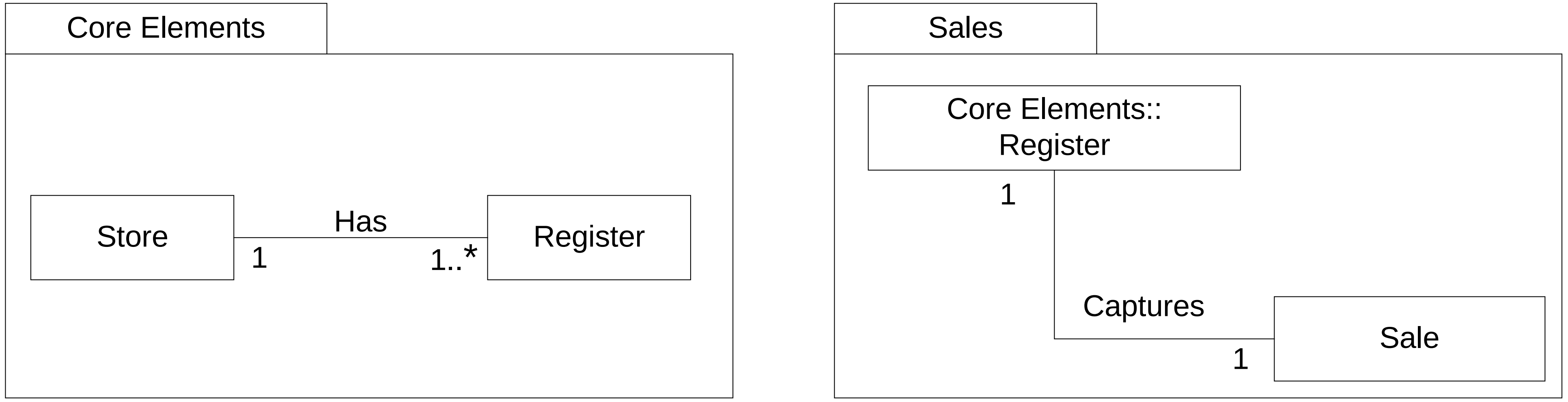
Often, a Domain Model may become complex and hard to read

Re-structuring the model using packages is an way to improve readability and maintainability



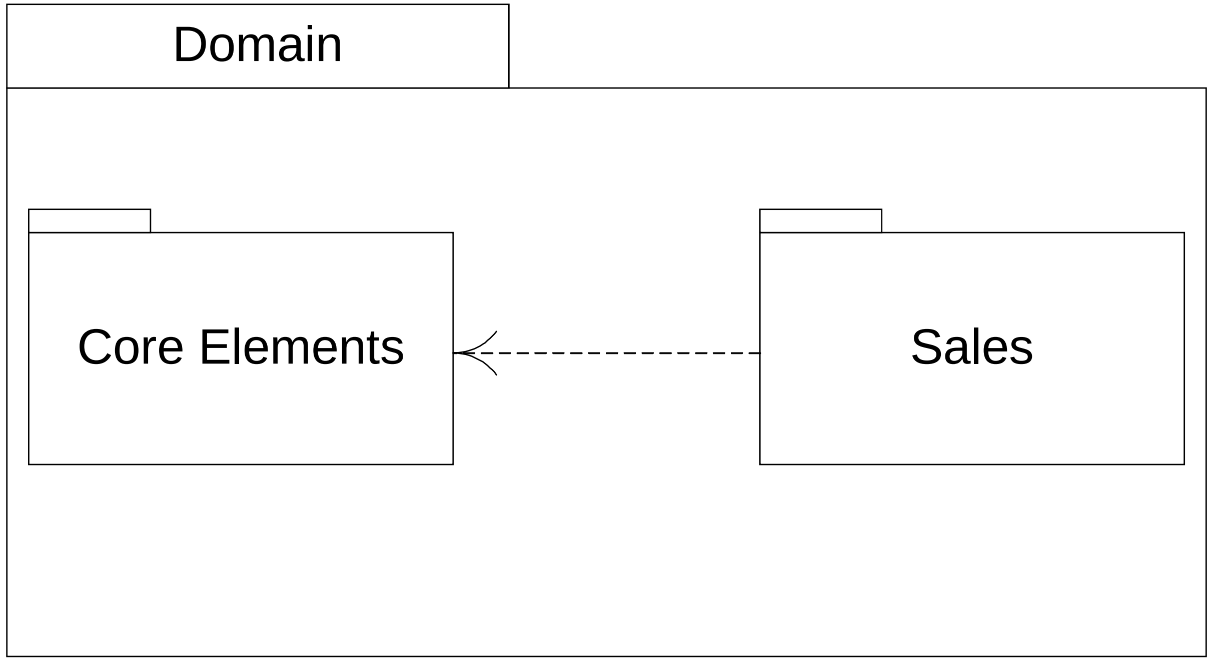
Packages

An element is owned by the package it belongs to, but can be referenced by elements in another package



Packages

Packages may be dependent upon each other; this happens if the elements know about or are coupled to elements in another package



Partitioning the Domain Model

Use packages to partition the Domain Model when there are groups of elements that …

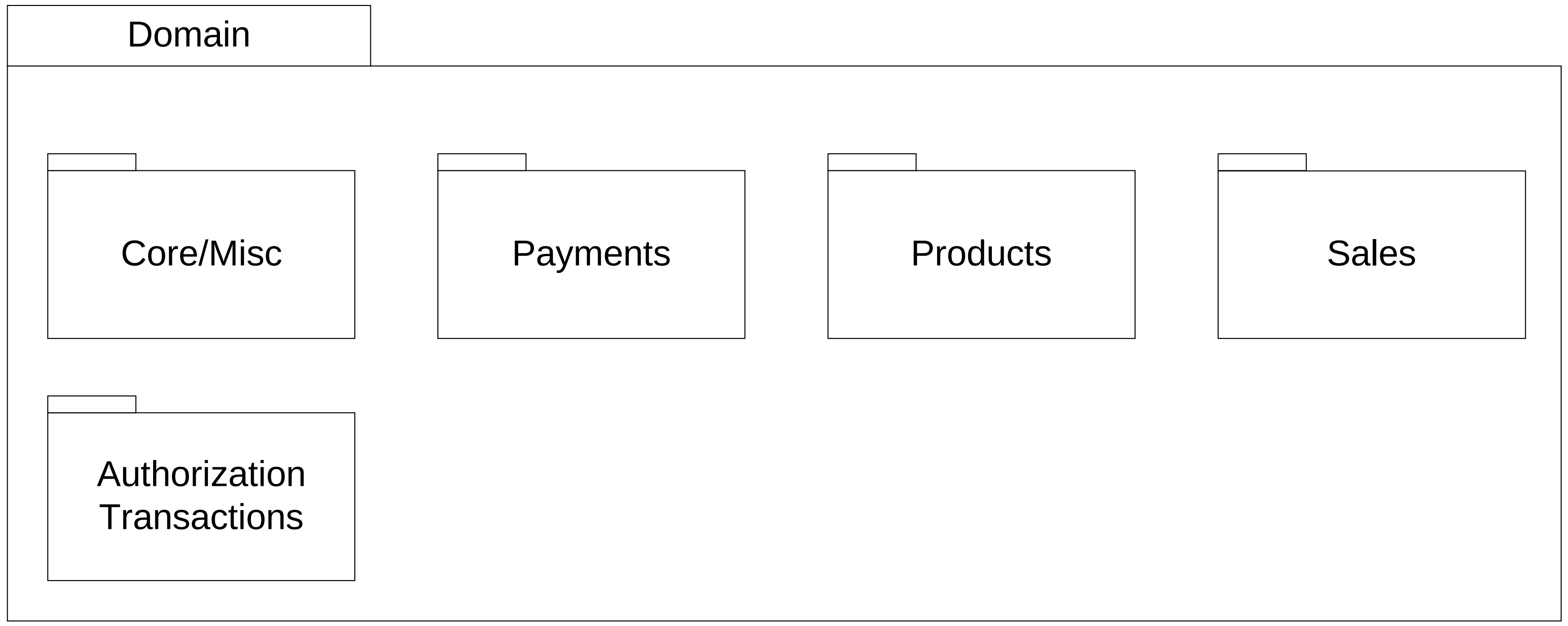
… are in the same subject area, closely related by concept or purpose

… are in the same class hierarchy together

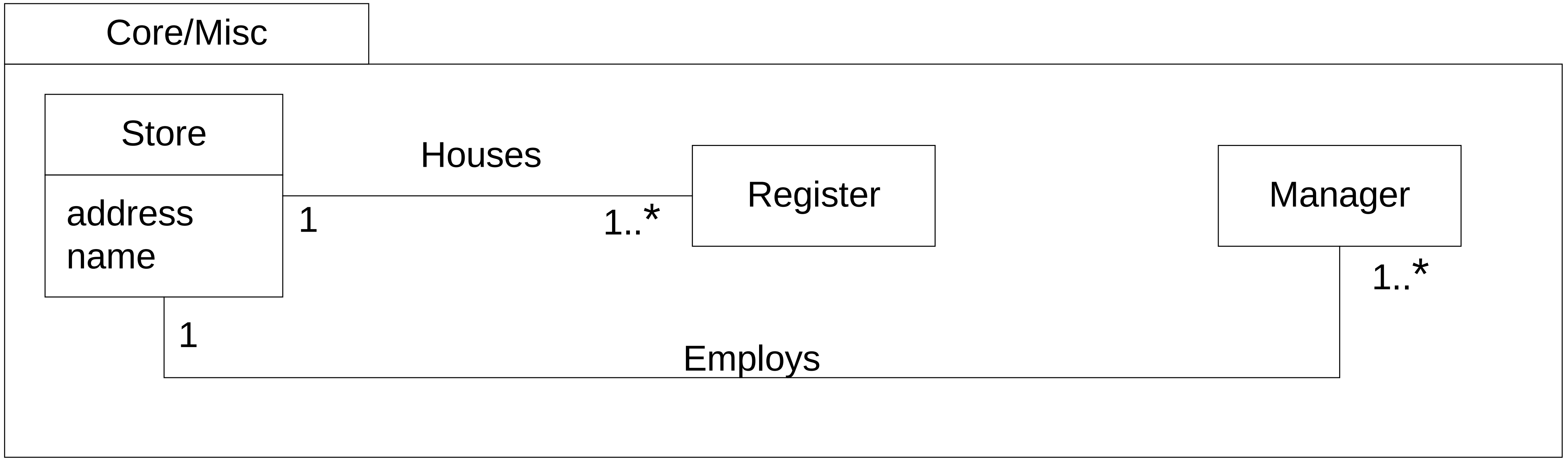
… participate in the same use case

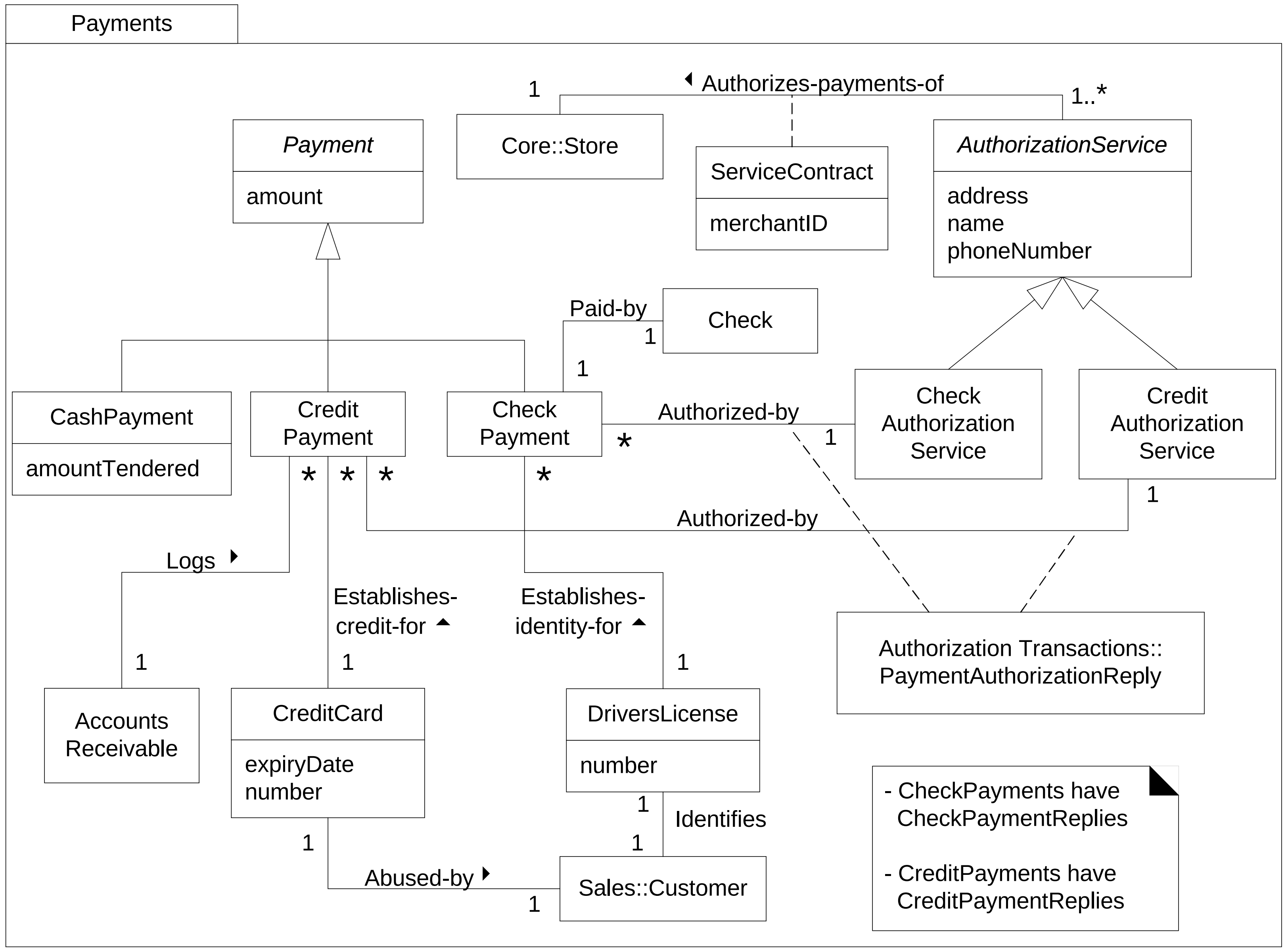
… are strongly associated

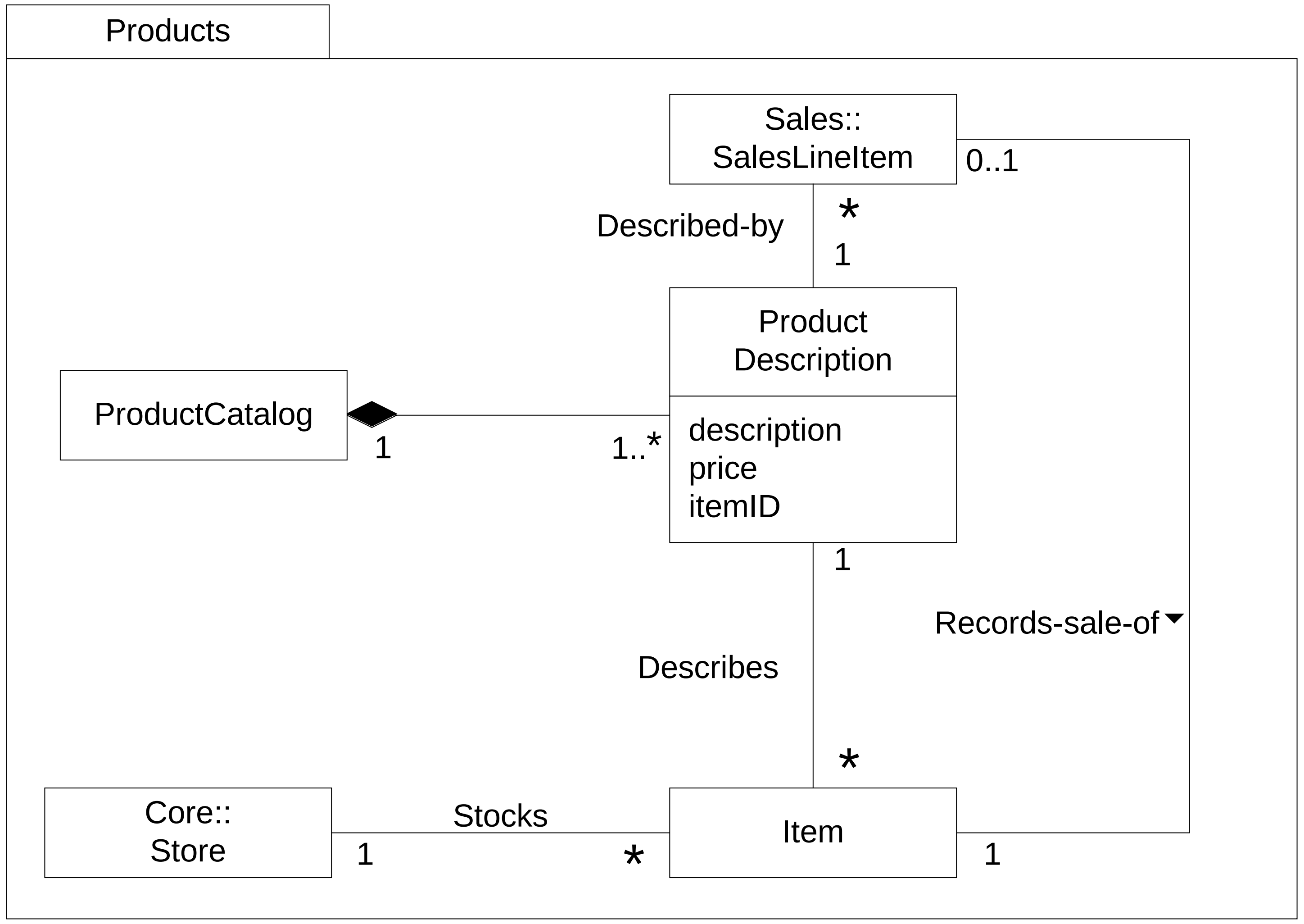
NextGen POS – Domain Model

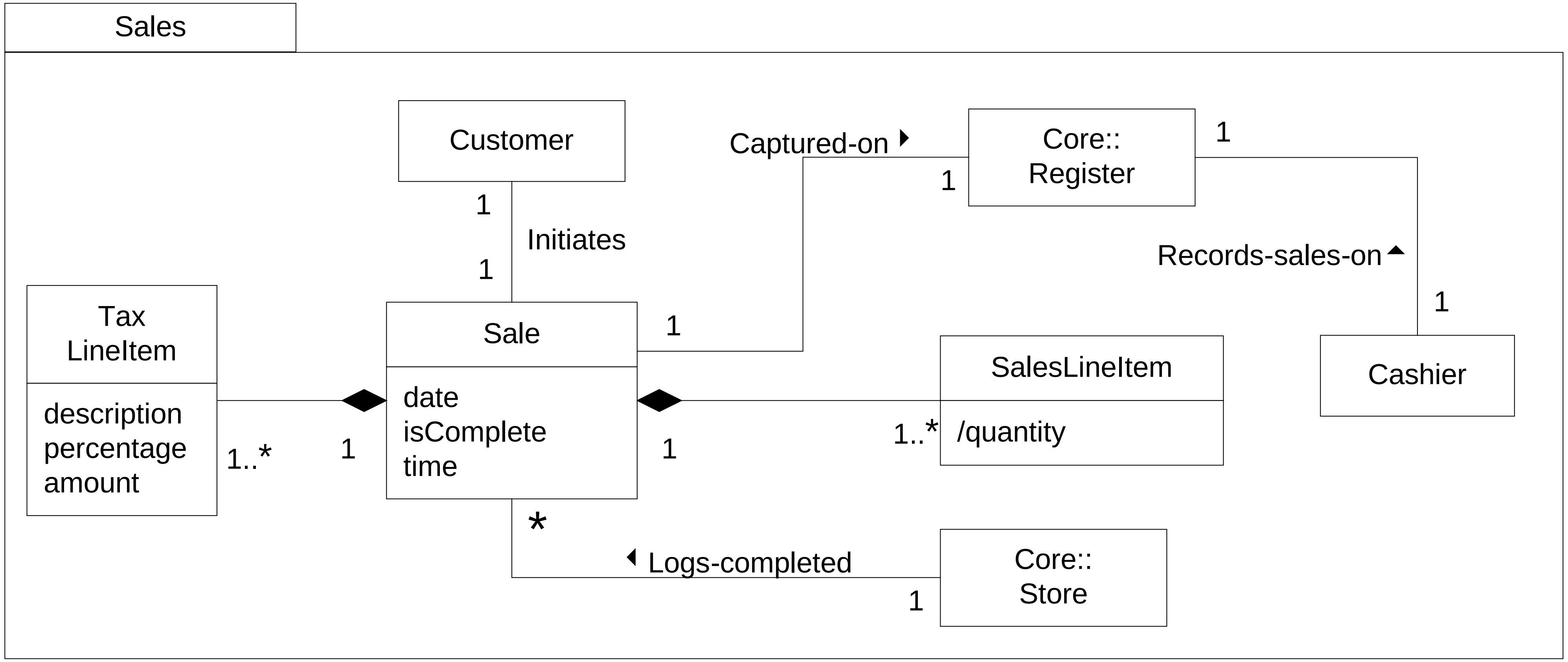


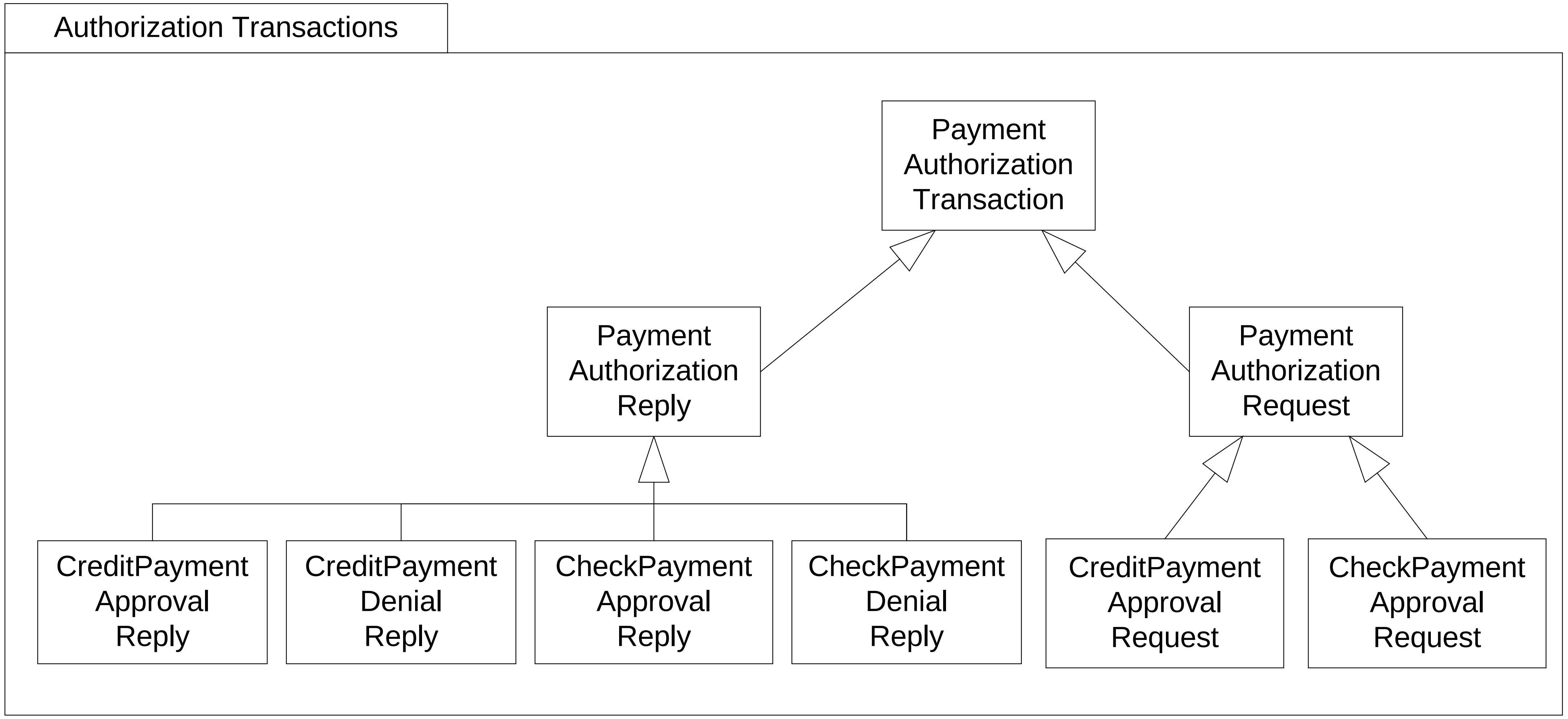
NextGen POS: Core/Misc Package




Payments
Payment
amount
Core::Store
ServiceContract
merchantID
AuthorizationService
address
name
phoneNumber
Authorizes-payments-of
1
1..*
Paid-by
Check
1
1
CashPayment
amountTendered
Credit Payment
Check Payment
Authorized-by
*
1
Check Authorization Service
Credit Authorization Service
*
*
*
*
1
Authorized-by
Logs
Establishes-credit-for
Establishes-identity-for
1
1
1
Accounts Receivable
CreditCard
expiryDate
number
DriversLicense
number
Authorization Transactions:: PaymentAuthorizationReply
1
Identifies
1
1
Abused-by
1
Sales::Customer
- CheckPayments have CheckPaymentReplies
- CreditPayments have CreditPaymentReplies


Products
Sales::
SalesLineItem
0..1
*
Described-by
1
Product
Description
description
price
itemID
ProductCatalog
1
1..*
1
Records-sale-of
Describes
*
Core::
Store
Stocks
1
*
Item
1


Sales
Customer
1
Initiates
1
Captured-on
1
Core:: Register
1
Records-sales-on
1
Cashier
Tax LineItem
description
percentage
amount
1..*
1
Sale
date
isComplete
time
1
1
SalesLineItem
/quantity
1..*
*
Logs-completed
1
Core:: Store


Authorization Transactions
Payment Authorization Transaction
Payment Authorization Reply
Payment Authorization Request
CreditPayment Approval Reply
CreditPayment Denial Reply
CheckPayment Approval Reply
CheckPayment Denial Reply
CreditPayment Approval Request
CheckPayment Approval Request