Principles of Software Construction: Objects, Design, and Concurrency (Part 2: Designing (Sub-)Systems)

What to build?

Jonathan Aldrich Charlie Garrod

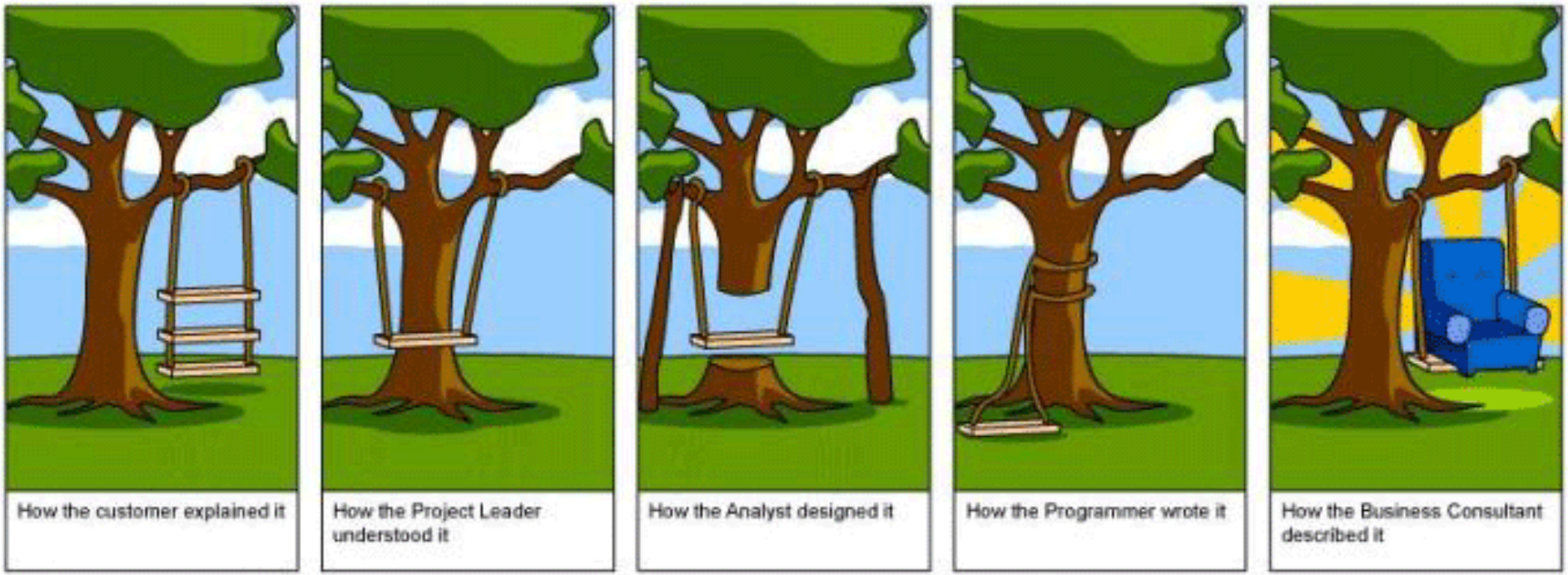
Learning Goals

- High-level understanding of requirements challenges
- Identify the key abstractions in a domain, model them as a domain model
- Identify the key interactions within a system, model them as system sequence diagram
- Discuss benefits and limitations of the design principle *"low representational gap"*

Design Goals, Principles, and Patterns

- Design Goals
  - Design for change
  - Design for division of labor
  - Design for reuse
- Design Principle
  - Low representational gap

REQUIREMENTS


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Requirements say what the system will do (and not how it will do it).

- *The hardest single part of building a software system is deciding precisely **what to build**.*
- *No other part of the conceptual work is as difficult as establishing the detailed technical requirements ...*
- *No other part of the work so cripples the resulting system if done wrong.*
- *No other part is as difficult to rectify later.*

— Fred Brooks

Requirements

- What does the customer want?
- What is required, desired, not necessary? Legal, policy constraints?
- Customers often do not know what they really want; vague, biased by what they see; change their mind; get new ideas…
- Difficult to define requirements precisely
- (Are we building the right thing? Not: Are we building the thing right?)

Human and social issues
15-313 topic

Lufthansa Flight 2904

- The Airbus A320-200 airplane has a software-based braking system that consists of:
  - Ground spoilers (wing plates extended to reduce lift)
  - Reverse thrusters
  - Wheel brakes on the main landing gear
- To engage the braking system, the **wheels of the plane must be on the ground**.

Lufthansa Flight 2904

There are two "on ground" conditions:

1. Both shock absorber bear a load of 6300 kgs
2. Both wheels turn at 72 knots (83 mph) or faster

- Ground spoilers activate for conditions 1 or 2
- Reverse thrust activates for condition 1 on both main landing gears
- Wheel brake activation depends upon condition 2

Requirements

- Wh
- Wh
  Leg
- Cus ally want; vague, biased by what they see; change th
- D
- (A
  b

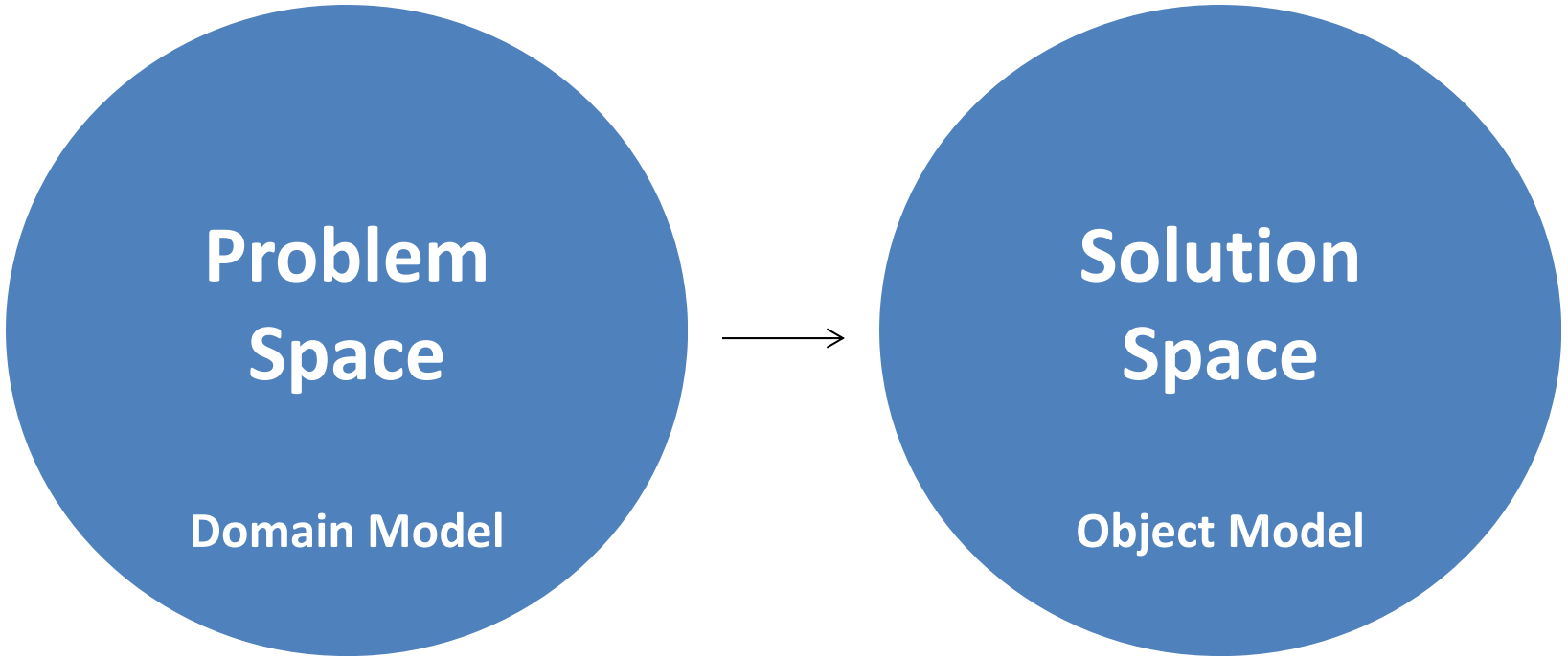
214 assumption:
Somebody has gathered the requirements (mostly text).

Challenges:
How do we start implementing them?
How do we cope with changes?

Human and social issue
15-313 topic

This lecture

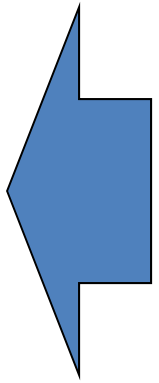
- Understand functional requirements
- Understand the problem's vocabulary (domain model)
- Understand the intended behavior (system sequence diagrams; contracts)



- Real-world concepts
- Requirements, Concepts
- Relationships among concepts
- Solving a problem
- Building a vocabulary

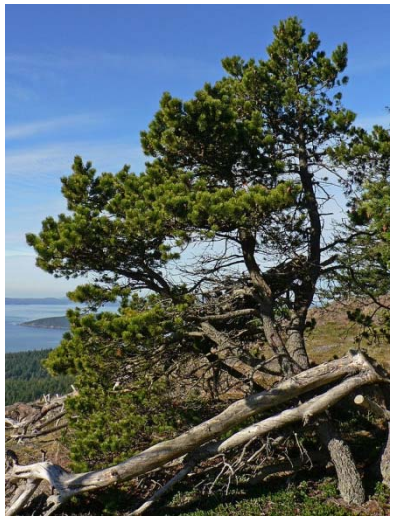
- System implementation
- Classes, objects
- References among objects and inheritance hierarchies
- Computing a result
- Finding a solution

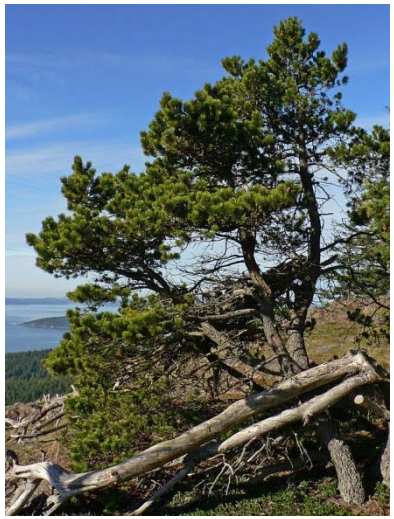
A design process

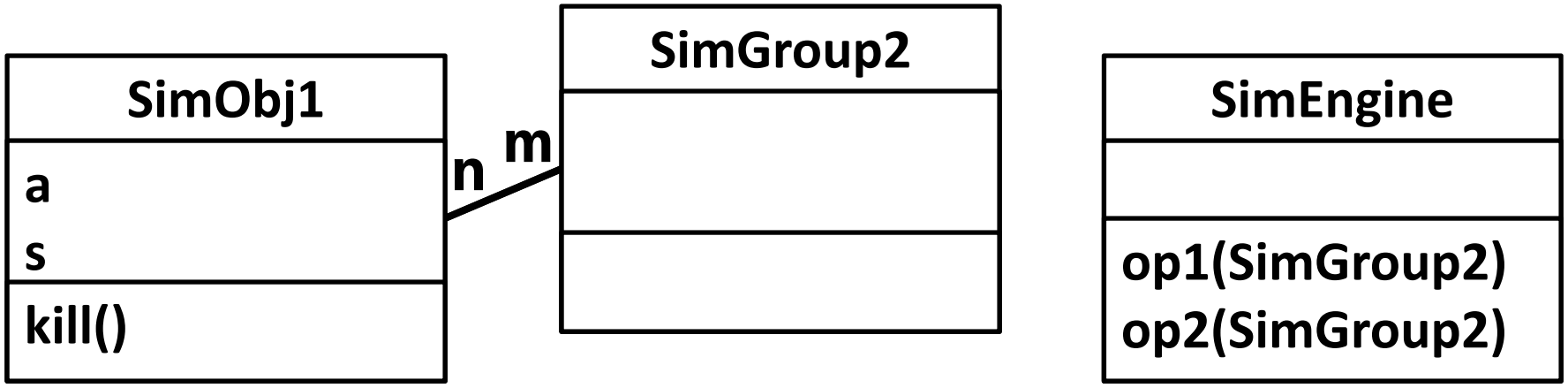


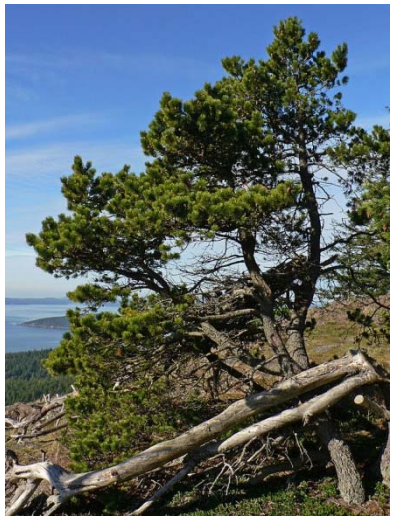
- Object-Oriented Analysis
  - Understand the problem
  - Identify the key concepts and their relationships
  - Build a (visual) vocabulary
  - Create a domain model (aka conceptual model)
- Object-Oriented Design
  - Identify software classes and their relationships with class diagrams
  - Assign responsibilities (attributes, methods)
  - Explore behavior with interaction diagrams
  - Explore design alternatives
  - Create an object model (aka design model and design class diagram) and interaction models
- Implementation
  - Map designs to code, implementing classes and methods

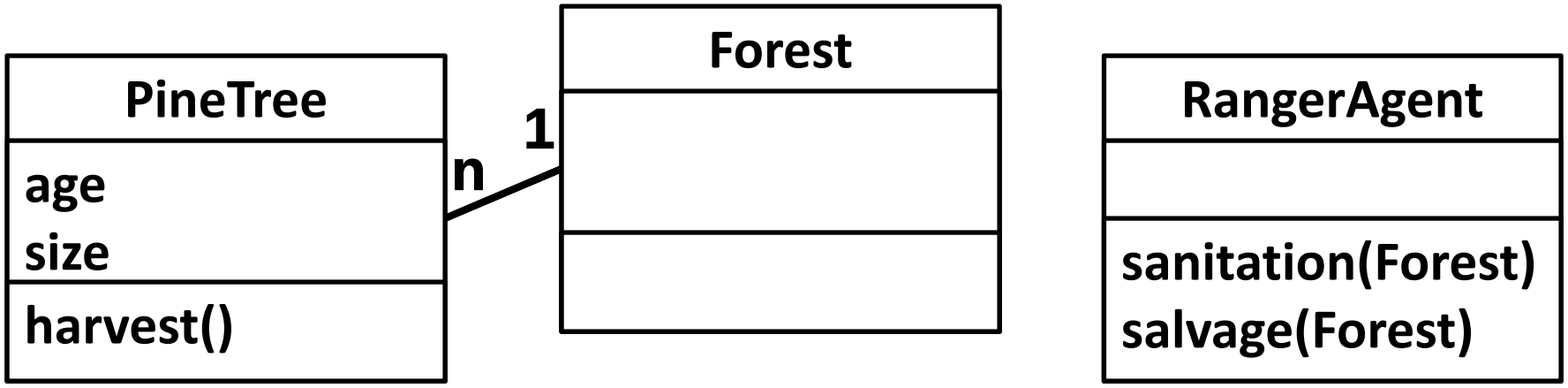
DESIGN PRINCIPLE:
LOW REPRESENTATIONAL GAP

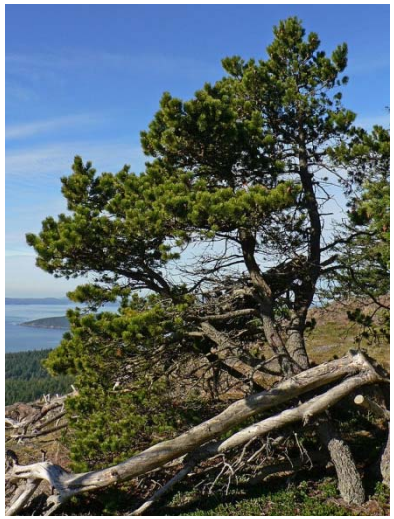


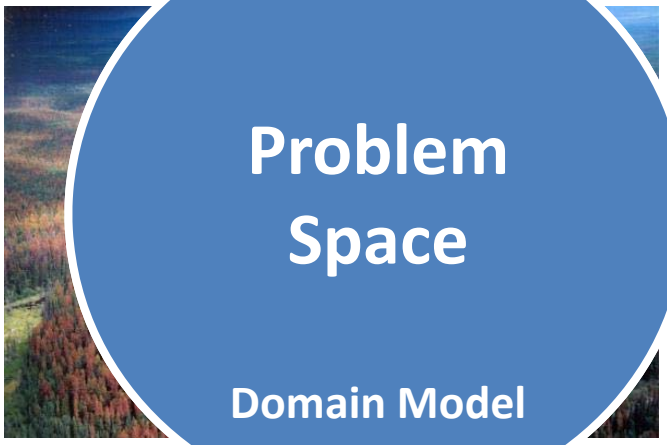



SimObj1
a
s
kill()
n
m
SimGroup2
SimEngine
op1(SimGroup2)
op2(SimGroup2)




PineTree
age
size
harvest()
n
1
Forest
RangerAgent
sanitation(Forest)
salvage(Forest)





Problem Space

Domain Model

inspires objects and names

PineTree
age size
harvest()

n — 1

Solution Space

Object Model

RangerAgent
sanitation(Forest) salvage(Forest)

Low Representational Gap (Congruency)

- Align software objects with real-world objects (concrete and abstract); objects with relationships and interactions in the real world similarly relate and interact in software
- "Intuitive" understanding; clear vocabulary
- Real-world abstractions are less likely to change => Design for change

=> Find and understand real-world objects and abstractions

Benefit of Low Representational Gap (Congruence)

- The domain model is familiar to domain experts
  - Simpler than code
  - Uses familiar names, relationships
- Classes in the object model and implementation will be inspired by domain model
  - similar names
  - possibly similar connections and responsibilities
- Facilitates understanding of design and implementation
- Facilitates traceability from problem to solution
- Facilitates evolution
  - Small changes in the domain more likely to lead to small changes in code

DOMAIN MODELS

Object-Oriented Analysis

- Find the concepts in the problem domain
  - Real-world abstractions, not necessarily software objects
- Understand the problem
- Establish a common vocabulary
- Common documentation, big picture
- For communication!
- Often using UML class diagrams as (informal) notation

- Starting point for finding classes later (low representational gap)

Why domain modeling?

- Understand the domain
  - Details matter! Does every student have exactly one major?
- Ensure completeness
  - A student's home college affects registration
- Agree on a common set of terms
  - freshman/sophomore vs. first-year/second-year
- Prepare to design
  - Domain concepts are good candidates for OO classes (-> low representational gap)
- A domain model is a (often visual) representation of the concepts and relationships in a domain

Running Example

Identify concepts

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

Running Example

- **Point of sale** (**POS**) or **checkout** is the place where a retail transaction is completed. It is the point at which a customer makes a payment to a merchant in exchange for goods or services. At the point of sale the merchant would use any of a range of possible methods to calculate the amount owing - such as a manual system, weighing machines, scanners or an electronic cash register. The merchant will usually provide hardware and options for use by the customer to make payment. The merchant will also normally issue a receipt for the transaction.
- For small and medium-sized retailers, the POS will be customized by retail industry as different industries have different needs. For example, a grocery or candy store will need a scale at the point of sale, while bars and restaurants will need to customize the item sold when a customer has a special meal or drink request. The modern point of sale will also include advanced functionalities to cater to different verticals, such as inventory, CRM, financials, warehousing, and so on, all built into the POS software. Prior to the modern POS, all of these functions were done independently and required the manual re-keying of information, which resulted in a lot of errors.

http://en.wikipedia.org/wiki/Point_of_sale

Read description carefully, look for nouns and verbs

- **Point of sale** (**POS**) or **checkout** is the place where a retail transaction is *completed*. It is the point at which a customer makes a payment to a merchant in *exchange* for goods or services. At the point of sale the merchant would use any of a range of possible methods to *calculate* the amount owing - such as a manual system, weighing machines, scanners or an electronic cash register. The merchant will usually provide hardware and options for use by the customer to *make* payment. The merchant will also normally *issue* a receipt for the transaction.
- For small and medium-sized retailers, the POS will be customized by retail industry as different industries have different needs. For example, a grocery or candy store will need a scale at the point of sale, while bars and restaurants will need to *customize* the item sold when a customer has a special meal or drink request. The modern point of sale will also include advanced functionalities to cater to different verticals, such as inventory, CRM, financials, warehousing, and so on, all built into the POS software. Prior to the modern POS, all of these functions were done independently and required the manual re-keying of information, which resulted in a lot of errors.

http://en.wikipedia.org/wiki/Point_of_sale

Hints for Identifying Concepts

- Read the requirements description, look for nouns
- Reuse existing models
- Use a category list
  - tangible things: cars, telemetry data, terminals, …
  - roles: mother, teacher, researcher
  - events: landing, purchase, request
  - interactions: loan, meeting, intersection, …
  - structure, devices, organizational units, …
- Analyze typical use scenarios, analyze behavior
- Brainstorming

- Collect first; organize, filter, and revise later

Identifying Relevant Concepts

- The domain model should contain only relevant concepts
- Remove concepts irrelevant for the problem
- Remove vague concepts (e.g., "system")
- Remove redundant concepts, agree on name
- Remove implementation constructs
- Distinguish attributes (strings, numbers) and concepts
- Distinguish operations and concepts

Identifying Concepts for the Library System

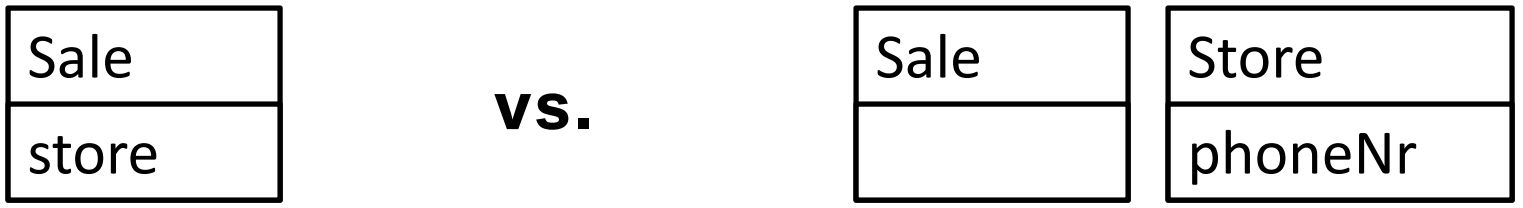
- Let's identify some concepts

- Library Scenario (sometimes called a Use Case)
  - User arrives at automated checkout system with books to borrow
  - User logs in with ID card
  - User scans books
  - System assigns due date
  - System prints receipt

Organize Concepts

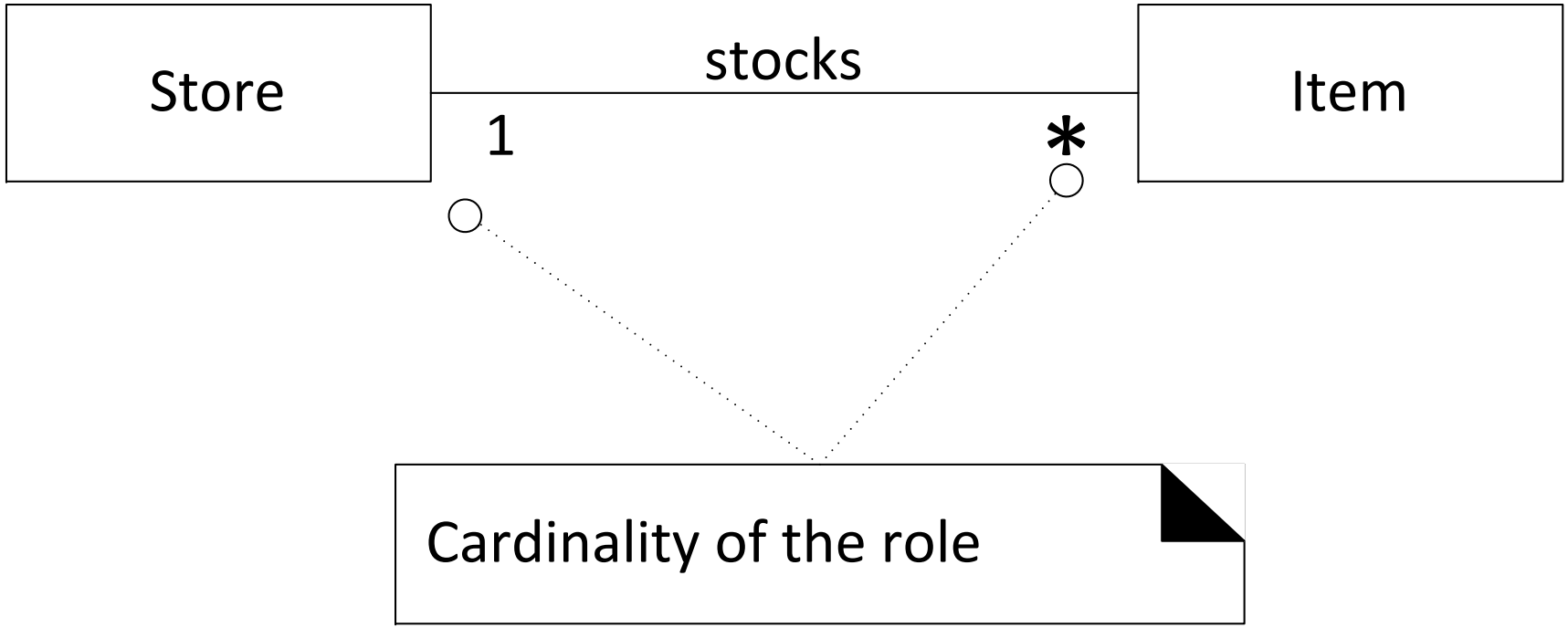
- Identify related elements
- Model relationships through inheritance ("is a") or associations ("related to")

Reminder: Classes vs. Attributes



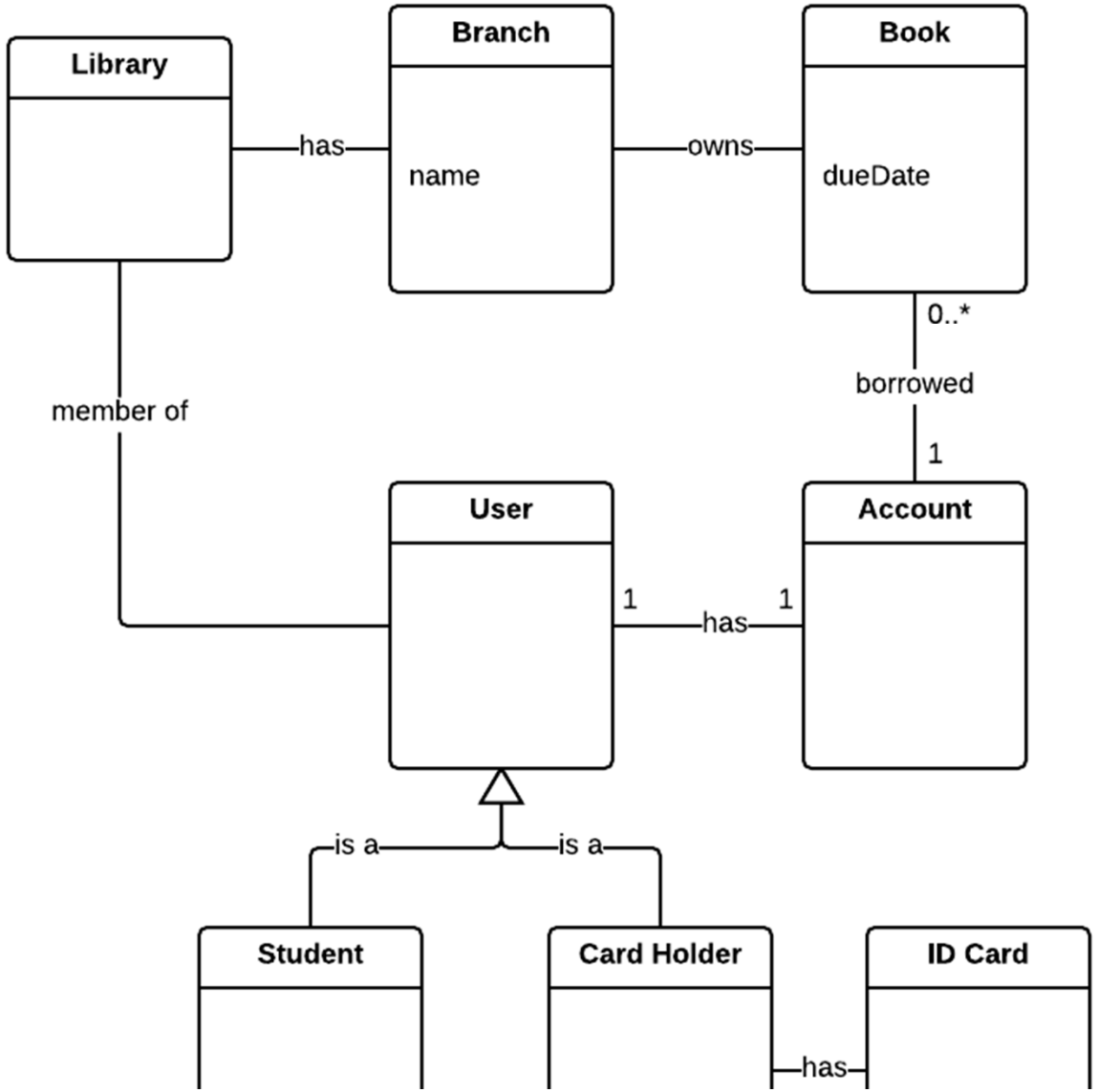
- "If we do not think of some conceptual class X as text or a number in the real world, it's probably a conceptual class, not an attribute"
- Avoid type annotations

Reminder: Associations



- When do we care about a relationship between two objects? (in the real world)
- Include cardinality (aka multiplicity) where relevant

Domain Model (example, excerpt)



Reminder: Lowering the Representational Gap (Congruency)

- Classes in the object model and implementation will be inspired by domain model
  - similar names
  - possibly similar connections and responsibilities
- Facilitates understanding of design and implementation
- Eases tracking and performing of changes

Hints for Object-Oriented Analysis

- A domain model provides vocabulary
  - for communication among developers, testers, clients, domain experts, …
  - Agree on a single vocabulary, visualize it
- Focus on concepts, not software classes, not data
  - ideas, things, objects
  - Give it a name, define it and give examples (symbol, intension, extension)
  - Add glossary
  - Some might be implemented as classes, other might not
- There are many choices
- The model will never be perfectly correct
  - that’s okay
  - start with a partial model, model what's needed
  - extend with additional information later
  - communicate changes clearly
  - otherwise danger of "analysis paralysis"

Documenting a Domain Model

- Typical: UML class diagram
  - Simple classes without methods and essential attributes only
  - Associations, inheritances, … as needed
  - Do not include implementation-specific details, e.g., types, method signatures
  - Include notes as needed
- Complement with examples, glossary, etc as needed
- Formality depends on size of project
- Expect revisions

Three perspectives on class diagrams

- Conceptual: Draw a diagram that represents the concepts in the domain under study
  - Conceptual classes reflect concepts in the domain
  - Little or no regard for software that might implement it
- Specification: Describing the interfaces of the software, not the implementation
  - Java interfaces
  - Java classes, but hiding information from a viewpoint
    - Private members – often use associations instead of fields
    - Hide other implementation choices not relevant for this view
- Implementation: Diagram describes code concretely, including implementation details

Domain Model Distinctions

- Vs. data model (solution space)
  - Not necessarily data to be stored
- Vs. object model and Java classes (solution space)
  - Only includes real domain concepts (real objects or real-world abstractions)
  - No “UI frame”, no database, etc.

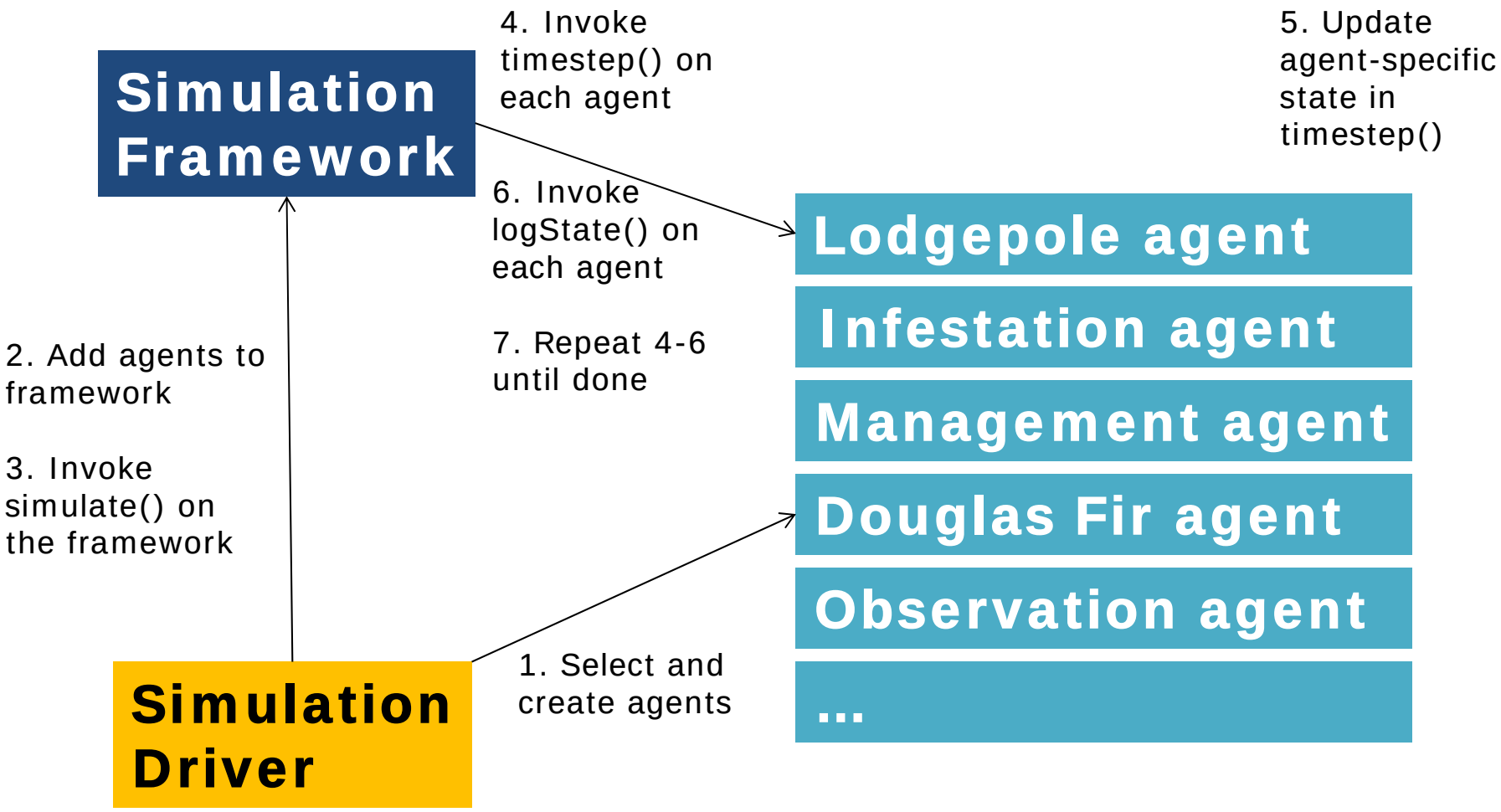
SYSTEM SEQUENCE DIAGRAMS

System Sequence Diagrams

- Domain model – understanding concepts and relationships in the domain
- What about interactions?
  - Between the program and its environment
  - Between major parts of the program

- A **System Sequence Diagram** is a model that shows, for one *scenario* of use, the sequence of events that occur on the system's boundary or between subsystems

Simulation Framework Behavior Model



(actually a Communication Diagram)

Interaction diagrams

- See textbook for notation of UML communication and sequence diagrams

Sequence vs Communication Diagrams

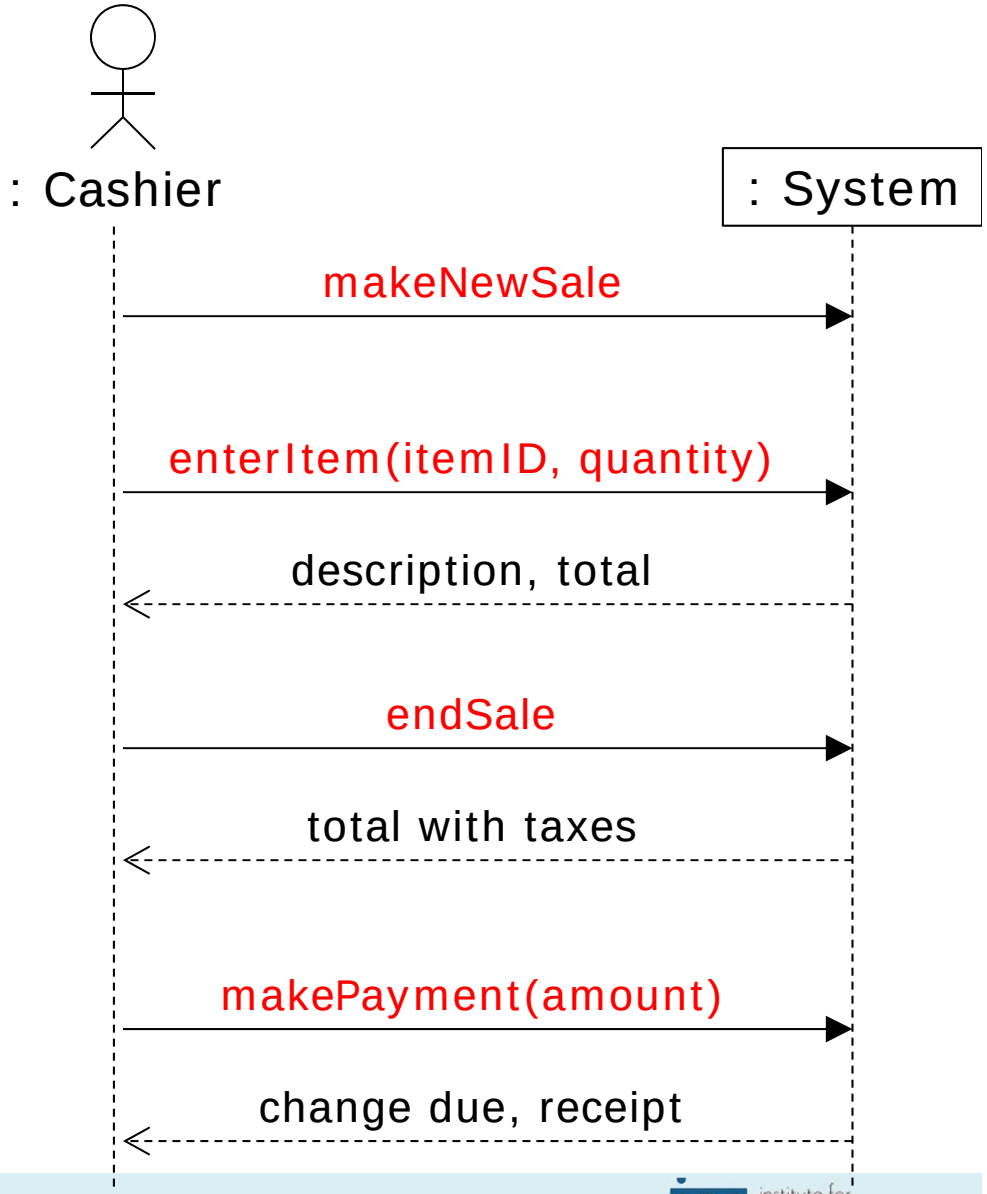
- Sequence diagrams are better to visualize the order in which things occur
- Communication diagrams also illustrate how objects are statically connected
- Communication diagrams often are more compact
- You should generally use interaction diagrams when you want to look at the behavior of several objects within a single use case.

A System Sequence Diagram for Borrowing a Book

- Library Scenario (sometimes called a Use Case)
  - User arrives at automated checkout system with books to borrow
  - User logs in with ID card
  - User scans books
  - System assigns due date
  - System prints receipt

Behavioral Contracts: What do These Operations Do?

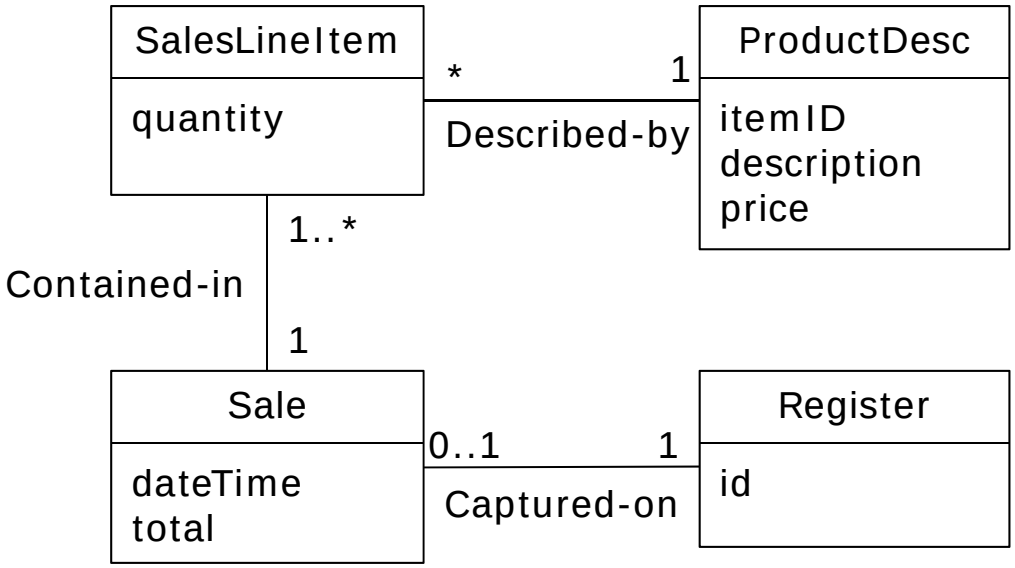
- Behavioral contract
  - Like a pre-/post-condition specification for code
  - Often written in natural language
  - **Focused on system interfaces**
    - may or may not be methods



Example Point of Sale Contract

Operation: makeNewSale()
Preconditions: none

Postconditions: - A Sale instance s was created
- s was associated with a Register



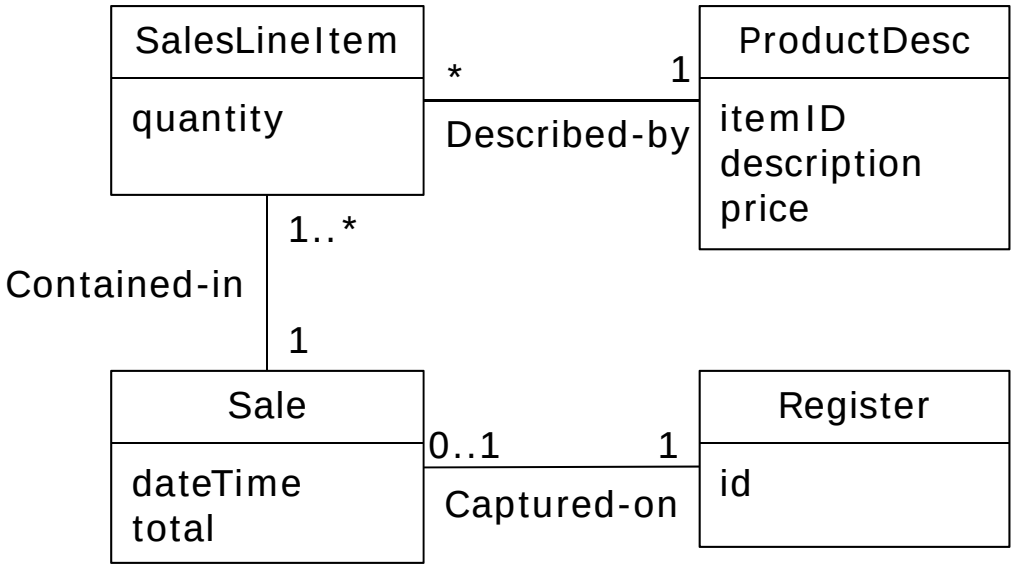
A Point of Sale Contract

- Contract structure
  - Operation name, parameters
  - Requirement or use case this is a part of
  - Preconditions
  - Postconditions
- Which contracts to write?
  - Operations that are complex or subtle
  - Operations that not everyone understands
  - Simple/obvious operations are often not given contracts in practice
- Writing postconditions
  - Written in past tense (a post-condition)
  - Describe changes to domain model
    - Instance creation and deletion
    - Attribute modification
    - Associations formed and broken
      - Easy to forget associations when creating objects!

Example Point of Sale Contract

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

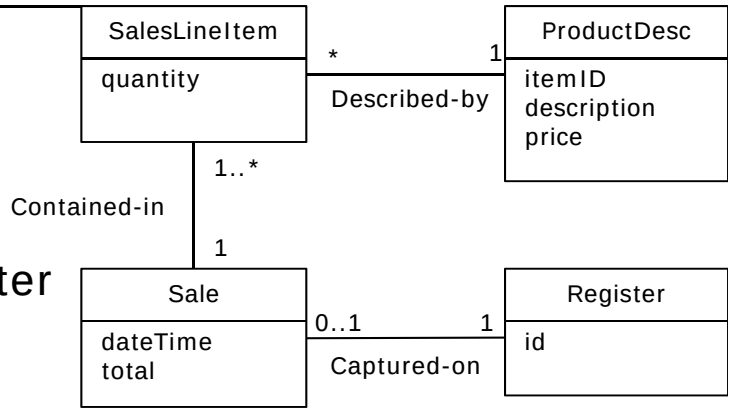
Postconditions:



Example Point of Sale Contracts

Operation: makeNewSale()
Preconditions: none

Postconditions: - A Sale instance s was created
- s was associated with a Register



Operation: enterItem(itemID : ItemID, quantity : integer)
Preconditions: There is a sale s underway

Postconditions: - A SalesLineItem instance sli was created
- sli was associated with the sale s
- sli.quantity became quantity
- sli was associated with a ProductDescription, based on itemID match

Take-Home Messages

- To design a solution, problem needs to be understood
- We assume requirements as given in this course, but need to understand them <- Domain modeling
- Domain models describe vocabulary and relationships of the problem space; useful to understand the domain
- System sequence diagrams model the interaction with the system, derive behavioral contracts
- Domain classes often turn into Java classes
  - Low representational gap principle to support design for understanding and change
  - Some domain classes don't need to be modeled in code; other concepts only live at the code level
- UML is a commonly understood notation for domain modeling; ensure right abstraction level