
# Principles of Software Construction: Objects, Design, and Concurrency

# Domain Modeling and Specifying System Behavior

***toad***

Fall 2014

**Jonathan Aldrich** Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

# Administrivia: Architecture Review Study

- What is the impact of structure architectural reviews on learning?
- To answer this question, later in the semester the TAs and some guests will review one of your assignments.
- We would like to use data from the reviews, assignments, tests, and quizzes.
  - All data will be anonymized
  - **Optional**. No impact on grading, profs/TAs will not know who gave consent.
- If you would like to participate:
  - Sign consent form
  - Get paid $10

# Review: From the Previous Lecture…

- What are some benefits of specifications?
- What is testing good for?
- How do we select good test cases?

# Exploring Continuous Integration

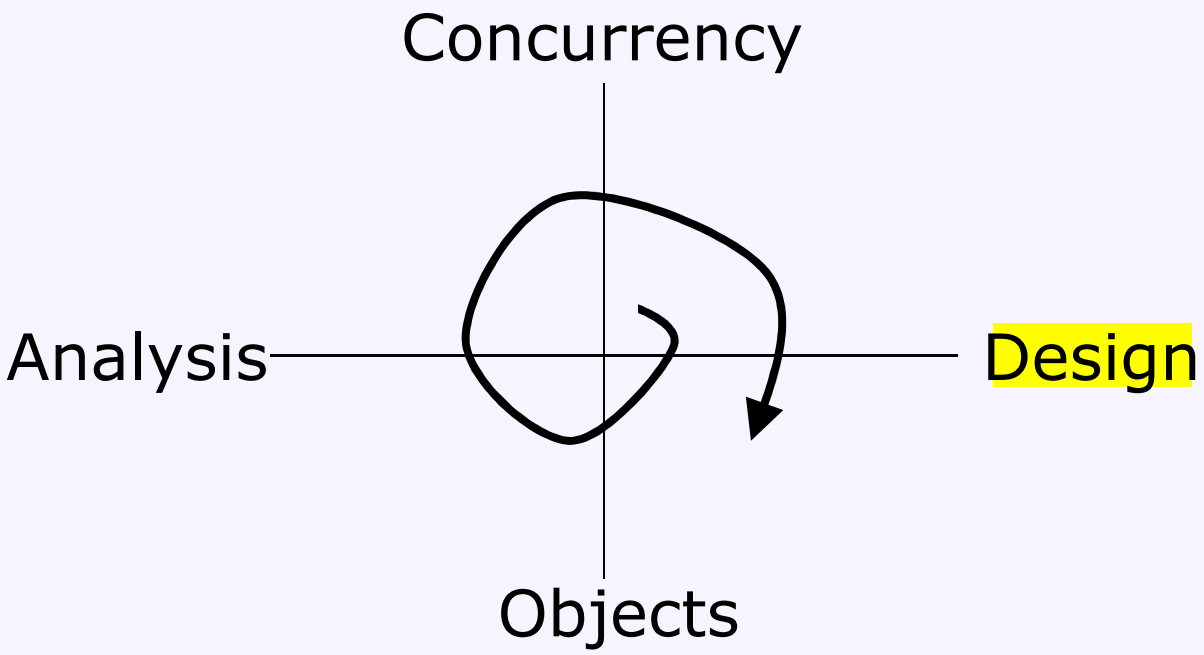
- When is continuous integration useful? Why not just test locally?

# Exploring Continuous Integration

- When is continuous integration useful? Why not just test locally?
  - In a large project, should enforce QA practices centrally
    - Otherwise there will always be someone who doesn't comply
  - There may be too many tests to run locally
    - Sheer volume of tests
    - Many configurations, environments
    - A CI server can run them all overnight
  - Want to standardize testing
    - Avoid any differences between developer machines

# This lecture

- 214: managing complexity, from programs to systems
  - **T**hreads and concurrency
  - **O**bject-oriented programming
  - **A**nalysis and modeling
  - **D**esign



# Steps in the Design Process

- Precondition: understand functional requirements
  - **Domain modeling**
  - System sequence diagrams
  - Behavioral contracts
- Precondition: understand quality attribute requirements
- Design a logical architecture
- Design a behavioral model
  - (Sub)system sequence diagrams
  - Behavioral contracts
- Responsibility assignment
- Interface design
- Algorithm and data structure design – pseudo-code

# Today's Lecture: Learning Goals

- Review specification, testing approaches, and continuous integration ✓
- Benefits of domain modeling
- Developing and documenting Domain Models
- Specifying program interactions with System Sequence Diagrams
- Specifying behavioral contracts for domain operations

# Domain Modeling

- Identify and relate the key concepts in a domain
  - Thus also called "conceptual modeling"
- Part of Object-Oriented Analysis
  - i.e. analysis of the problem space

- Why domain modeling?
  - You may not know the domain well
    - Details matter!  Does every student have exactly one major?
  - You don't want to forget key concepts
    - A student's home college affects registration
  - You want to 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

- A **domain model** is a visual representation of the concepts and relationships in a domain

# Domain Model Distinctions

- Vs. data model
  - Not necessarily data to be stored
- Vs. Java classes
  - Only includes real domain concepts
  - No “window” in the UI, no database, etc.

# Domain Modeling Steps

- Identify concepts
- Establish a common vocabulary
- Add associations between concepts
- Assign attributes to the concepts
- Find commonalities between concepts

# Running Example

# 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 Concepts for the Point of Sale Example

- Let's identify some concepts

- Point of Sale (POS) Scenario (sometimes called a *Use Case*)
  1. Customer arrives at POS checkout with goods 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
  5. Cashier repeats steps 3-4 until all goods have been entered
  6. System presents total with taxes calculated
  7. Cashier tells customer the total, and asks for payment
  8. Customer pays and System provides change and a receipt

# First Steps toward a Conceptual Model

## Identify concepts

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

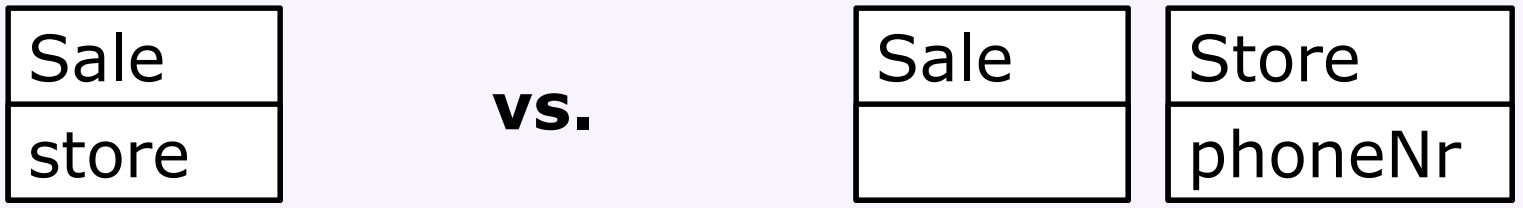
# Identifying Relevant Concepts

- 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
  - Pick whatever name is used in the real domain
- Remove implementation constructs
- Distinguish attributes (strings, numbers) and concepts
- Distinguish operations and concepts

# Organize Concepts

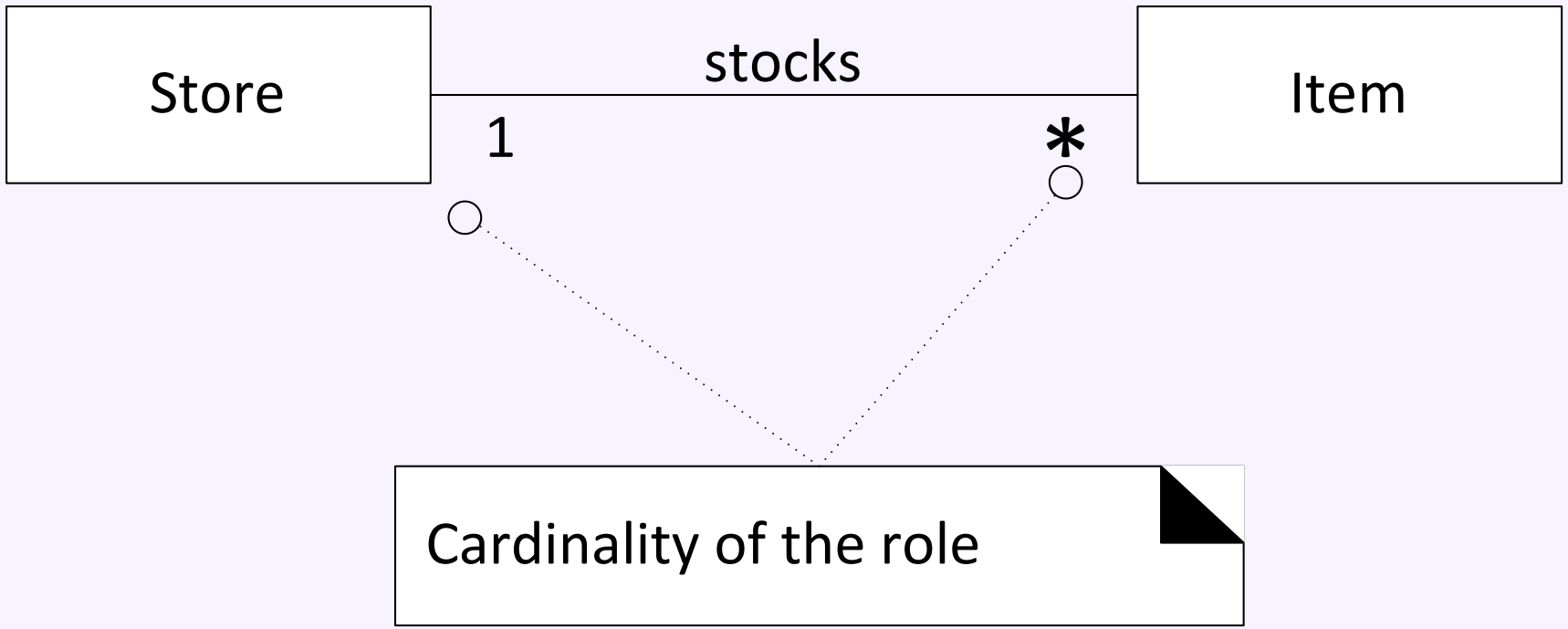
- Identify related elements
- Model relationships through specialization ("is a") or associations ("related to")

# Classes vs. Attributes



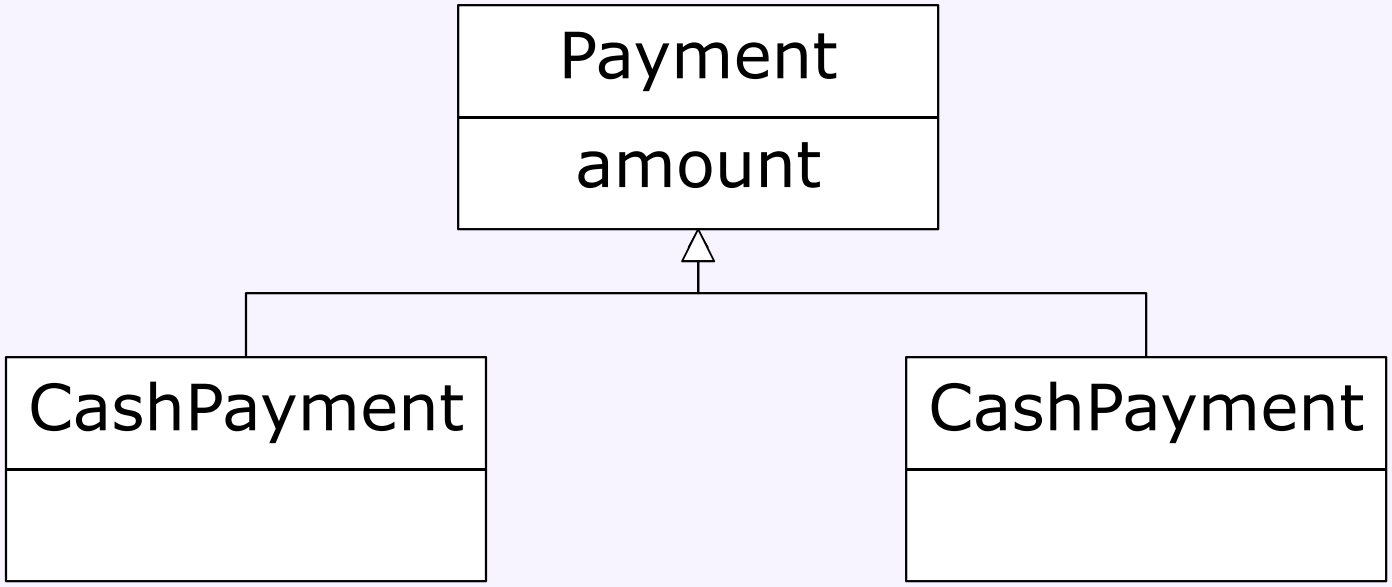

- "If we do not think of some conceptual class X as text or a number in the real world, it's probably a conceptual class, not an attribute"
- Avoid type annotations

# Associations




- When do we care about a relationship between two objects? (in the real world)
  - Guideline: is this a relationship that needs to be remembered?
- Name with a verb phrase, as in "X –verb phrase→ Y"
- Include cardinality (aka multiplicity) where relevant

# Specialization




- Sometimes several concepts are *specializations* of a more general concept
- We avoid repeating shared attributes and associations by putting them in the general concept
- Specialized concepts *inherit* these attributes and associations from their parent

# OO Analysis Benefit: 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
- Benefits of congruence
  - 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

# 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, inheritance, … 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

## Two uses for class diagrams

- **Domain modeling**: 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
- **Software design**: Diagram describes actual **implementation classes**

*Understanding the intended perspective is crucial to drawing and reading class diagrams*

# Associations

- Associations represent relationships between instances of classes
- Domain modeling: Associations represent conceptual relationships
- Software design: Associations represent pointers/fields between related classes

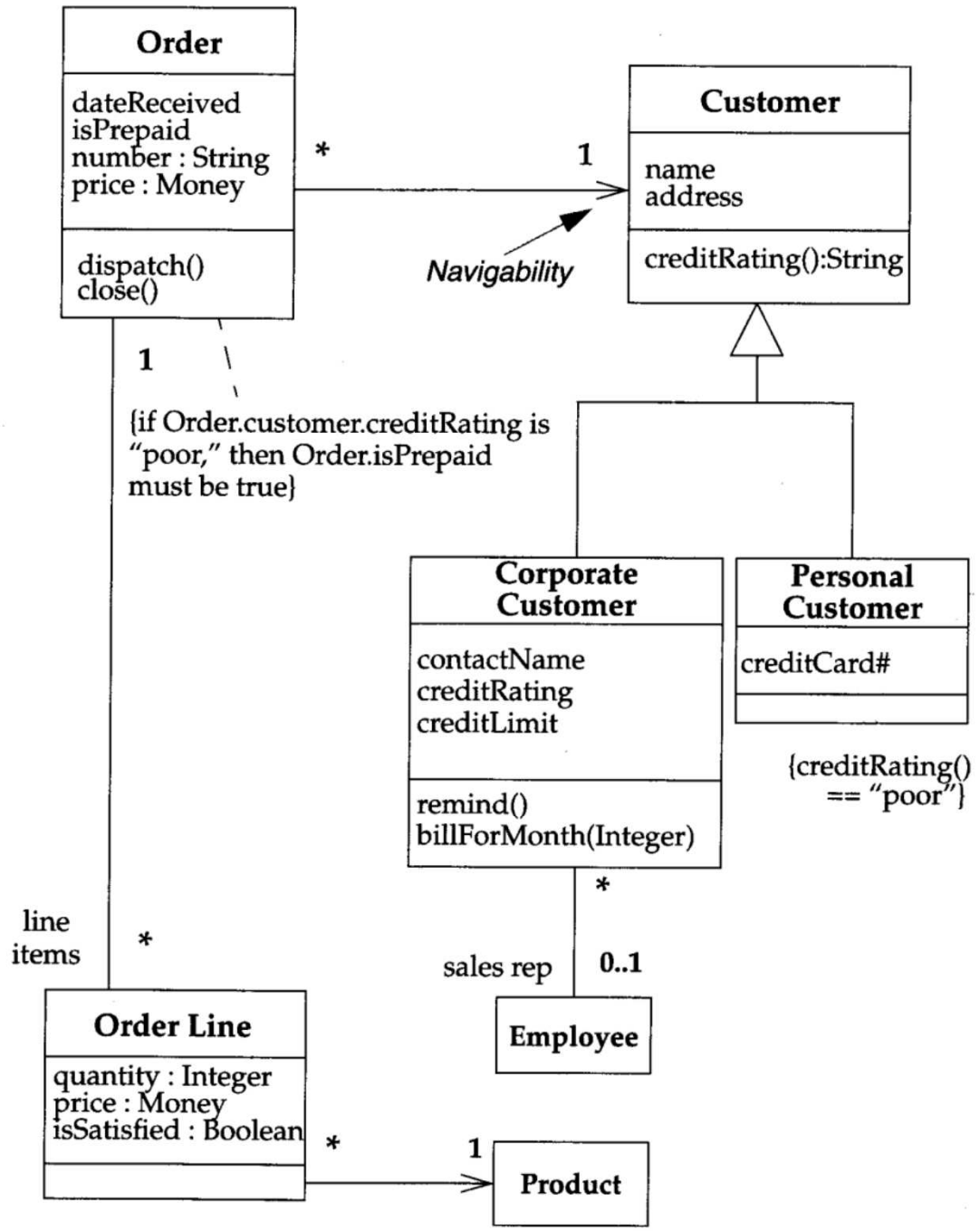
# Associations

- Each association has two ends
  - Each end can be named with a label called role name
  - An end also has a multiplicity: How many objects participate in the given relationship
    - General case: give upper and lower bound in lower..upper notation
    - Abbreviations: * = 0..infinity, 1 = 1..1
    - Most common multiplicities: 1, *, 0..1

## Associations
## Unidirectional vs bidirectional

- Arrows in association lines indicate navigability
  - Only one arrow: unidirectional association
  - No or two arrows: bidirectional association
- Software design: arrows indicate which objects contain a pointer to the other objects
- Arrows serve no useful purpose in domain modeling
- For bidirectional associations, the two navigations must be inverses of each other

# Unidirectional Associations



# Class Diagrams: Attributes

- Attributes are very similar to associations
  - Domain modeling: A customer's name attribute indicates that customers have names
  - Software design: customer has a field for its name
  - UML syntax for attributes:
    *visibility name : type = defaultValue*
    - Details may be omitted

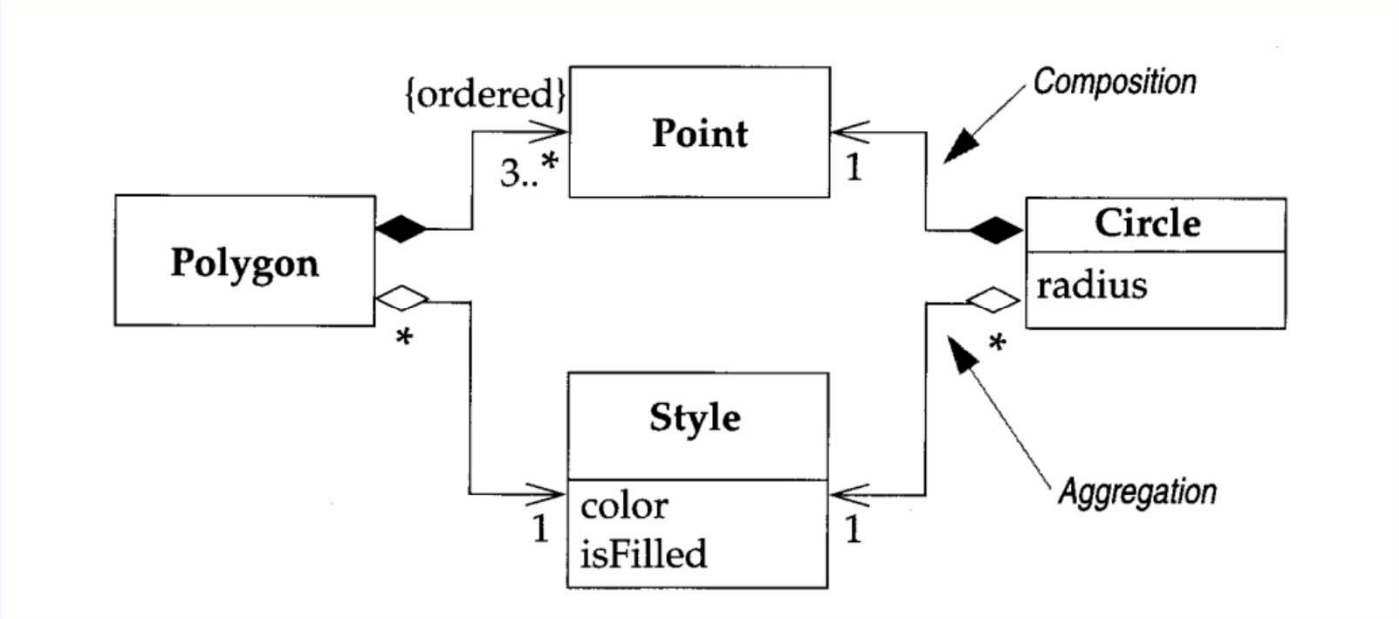
# Class Diagrams: Attributes vs Associations

- Attributes describe non-object-oriented data
  - Integers, strings, booleans, …
- For domain modeling this is the only difference
- Software design
  - Attributes imply navigability from type to attribute only
  - Implied that type contains solely its own copy of the attribute objects

# Class Diagrams: Operations

- Operations are the processes that a class knows to carry out
- Most obviously correspond to methods on a class
- Full syntax:
  *visibility name(parameter-list) : return-type*
  - *visibility* is + (public), # (protected), or - (private)
  - *name* is a string
  - *parameter-list* contains comma-separated parameters whose syntax is similar to that for attributes
    - Can also specificy direction: input (in), output(out), or both (inout)
    - Default: in
  - *return-type* is comma-separated list of return types (usually only one)

# Aggregation vs Composition

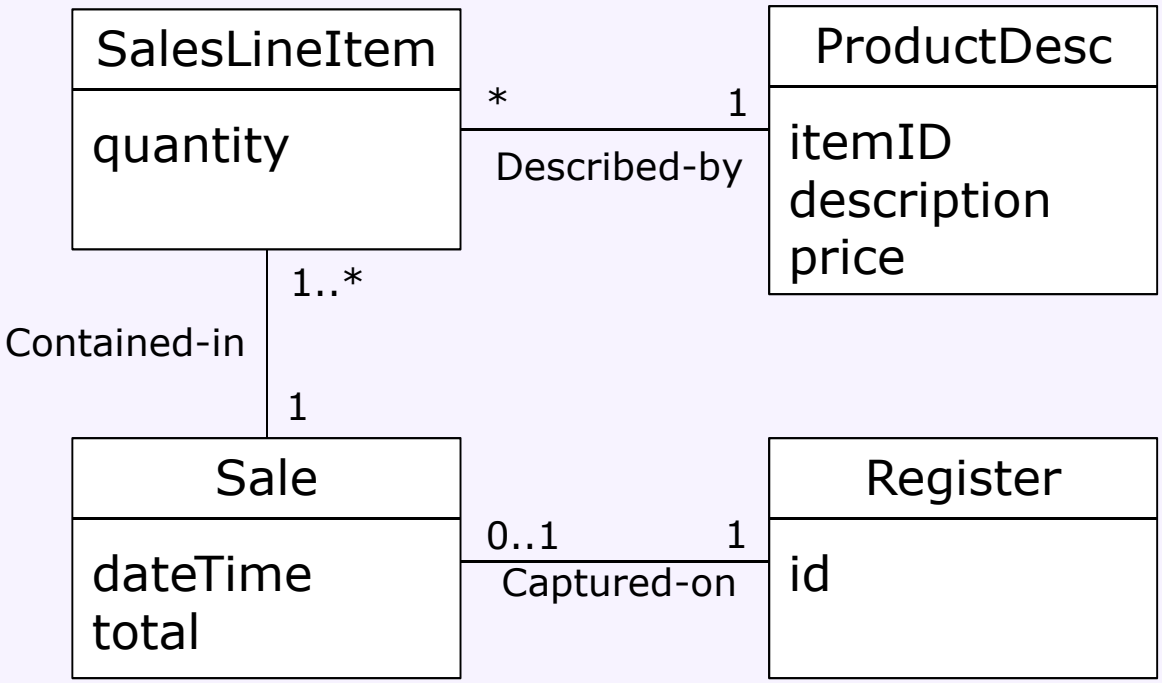



- Aggregation expresses "part-of" relationships, but rather vague semantics
- Composition is stronger: Part object live and die with the whole

# Build a UML Domain Model for the Point of Sale Scenario

- Point of Sale (POS) Scenario (sometimes called a *Use Case*)
  1. Customer arrives at POS checkout with goods 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
  5. Cashier repeats steps 3-4 until all goods have been entered
  6. System presents total with taxes calculated
  7. Cashier tells customer the total, and asks for payment
  8. Customer pays and System provides change and a receipt

# A Partial Domain Model



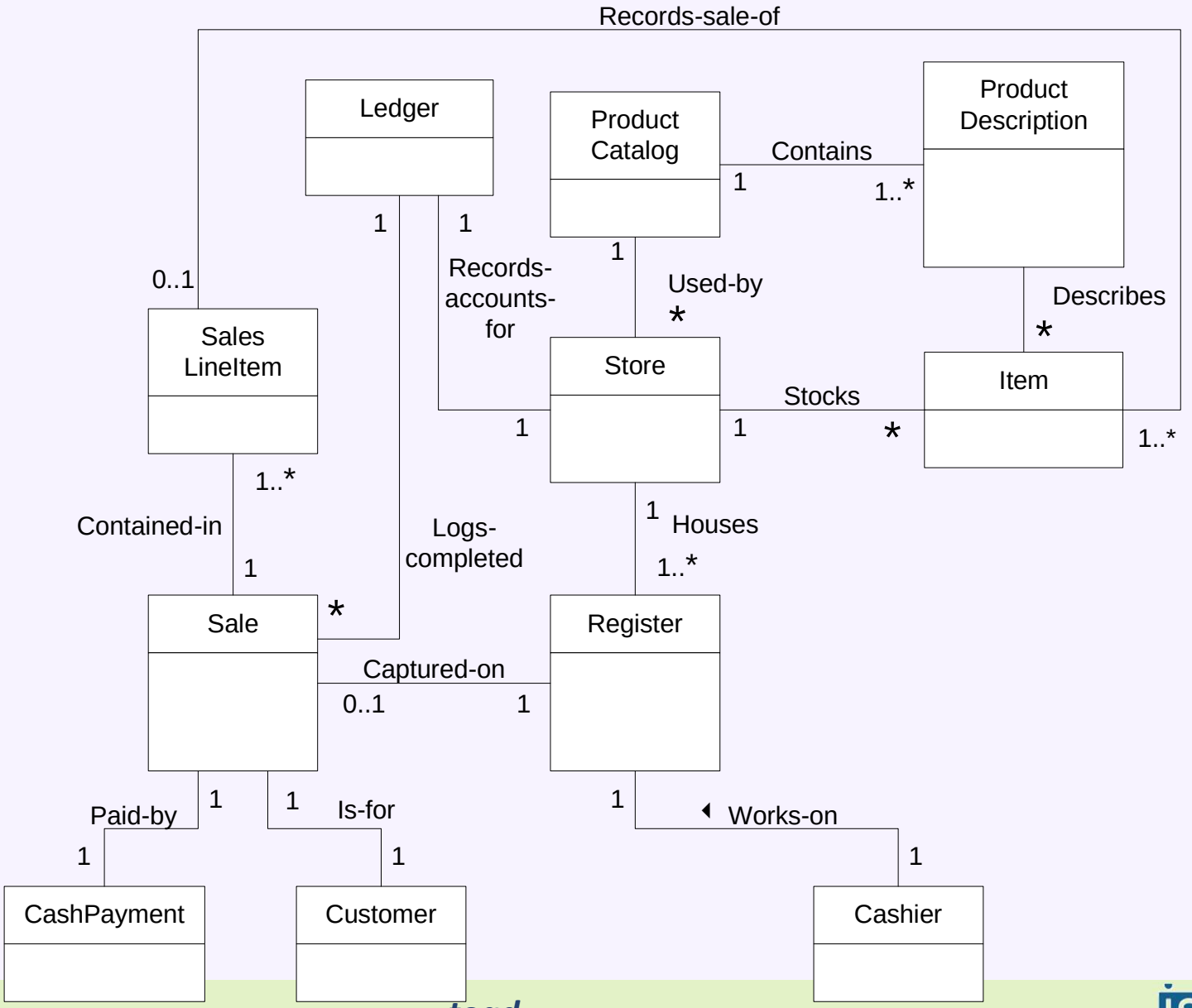
# A word on notation

- UML notation is broadly known, well documented
- Informal notations/sketching often sufficient, but potentially ambiguous for communication and documentation
- In practice:
  - Graphical modeling very common in general
  - Agree on some notation
  - Adapt/extend as needed
  - UML rarely full heartedly adopted
- In this course
  - Use UML and conventions for communication
  - Keep it simple
  - Clarity is imperative, document your extensions/shortcuts
  - We don't require or recommend a drawing tool

# Aside: Key Observations

- How used?
  - transient forms for exploration, permanent solutions for communication with larger groups
  - mostly ad-hoc white-board diagrams during meetings
- Why used?
  - to understand, to design, to communicate
  - "code is king"
- Graphical conventions?
  - Use of formal diagramming language is low
  - too formal for mostly informal visualizations; cost benefit ratio
- Culture?
  - limited adoption of drawing tools;
  - high value diagrams recreated more formally

# Domain Model (more complete, but without attributes)



# Steps in the Design Process

- Precondition: understand functional requirements
  - Domain modeling
  - **System sequence diagrams**
  - Behavioral contracts
- Precondition: understand quality attribute requirements
- Design a logical architecture
- Design a behavioral model
  - **(Sub)system sequence diagrams**
  - Behavioral contracts
- Responsibility assignment
- Interface design
- Algorithm and data structure design – pseudo-code

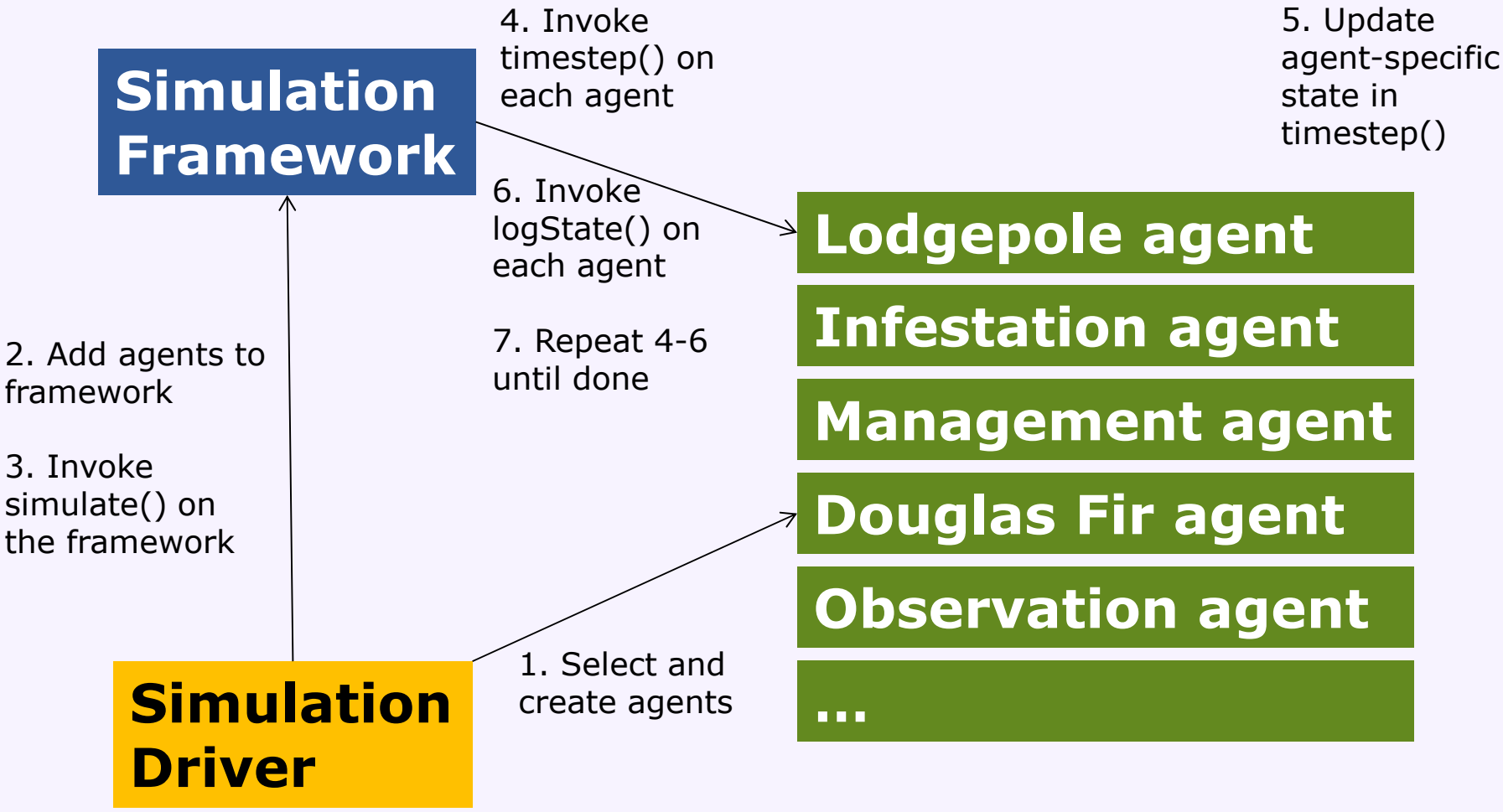
# System Sequence Diagrams

- 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 picture 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

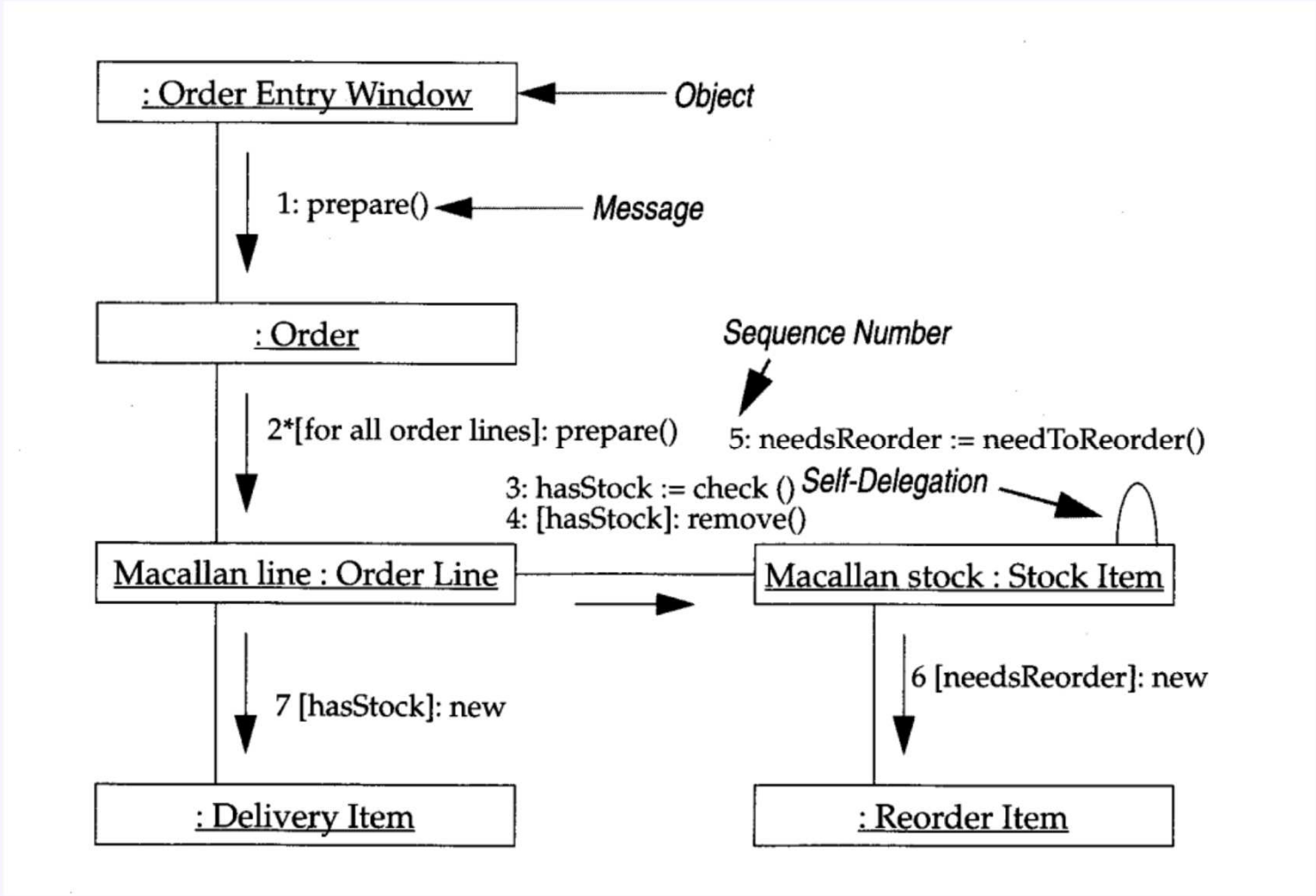
(actually a Communication Diagram)



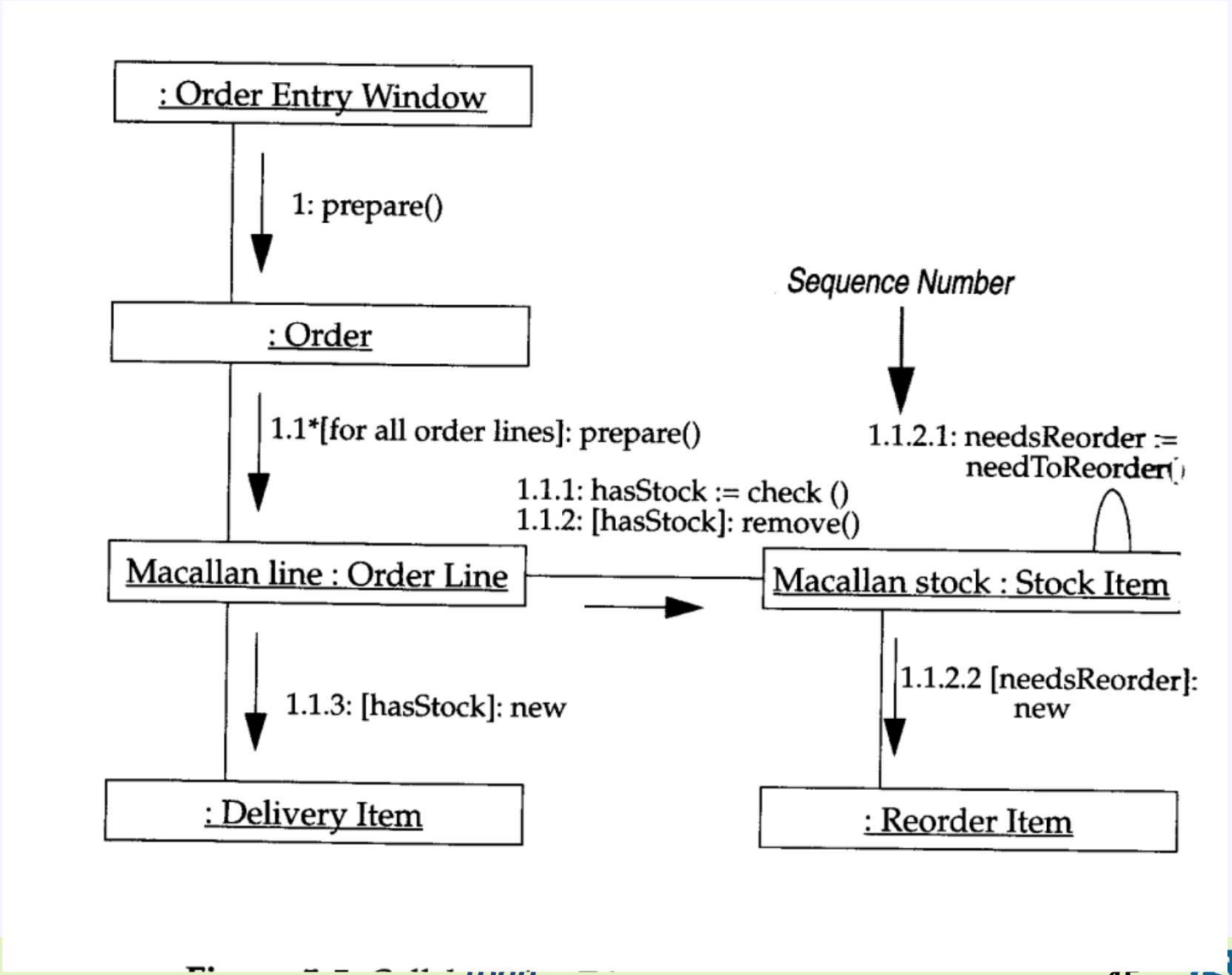
# Interaction Diagrams

- Interaction diagrams describe how groups of objects collaborate in some behavior
- Two kinds of interaction diagrams: **sequence diagrams** and **communication diagrams**

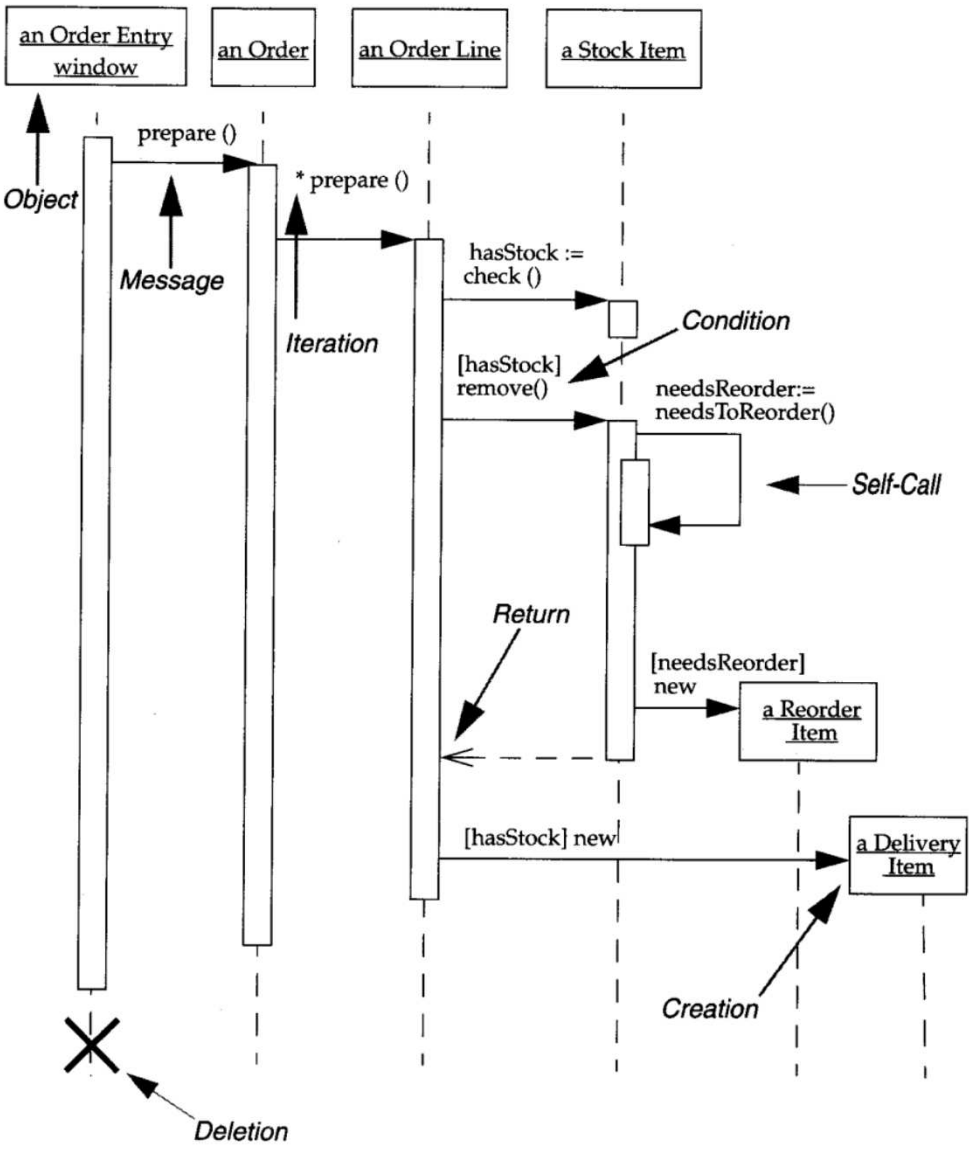
# Communication Diagram Example



# Communication Diagram Example: Decimal Numbering System



# Sequence Diagram Example



# Sequence Diagrams

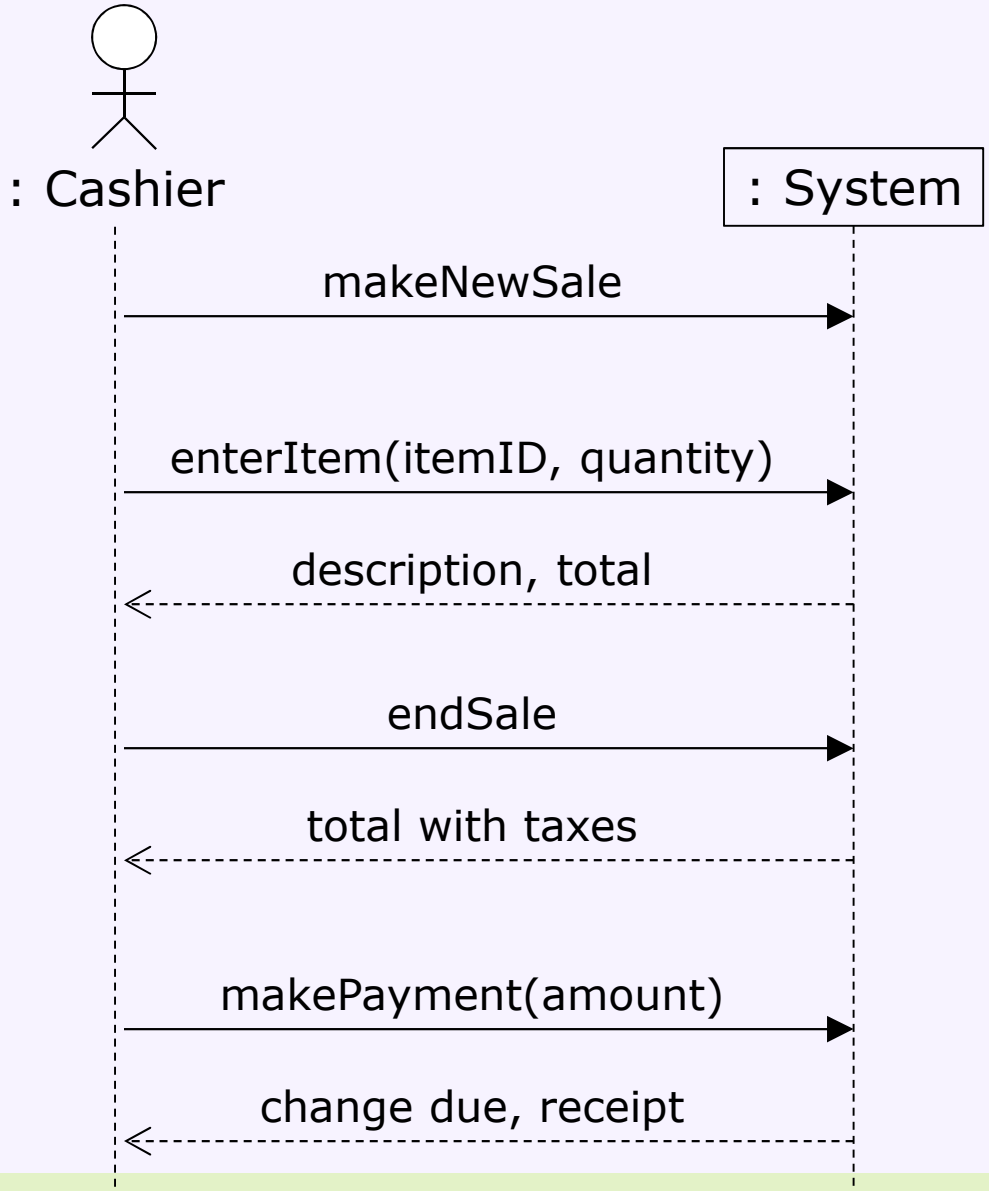
- Vertical line is called lifeline
- Each message represented by an arrow between lifelines
  - Labeled at minimum with message name
  - Can also include arguments and control information
  - Can show self-call by sending the message arrow back to the same lifeline
- Can add condition which indicates when message is sent, such as [needsReorder]
- Can add iteration marker which shows that a message is sent many times to multiple receiver objects

# Sequence Diagram for the Point of Sale Example

- Let's develop a sequence diagram

- Point of Sale (POS) Scenario (sometimes called a *Use Case*)
  1. Customer arrives at POS checkout with goods 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
  5. Cashier repeats steps 3-4 until all goods have been entered
  6. System presents total with taxes calculated
  7. Cashier tells customer the total, and asks for payment
  8. Customer pays and System provides change and a receipt

# Sequence Diagram for the Point of Sale Example



# Sequence vs Communication Diagrams

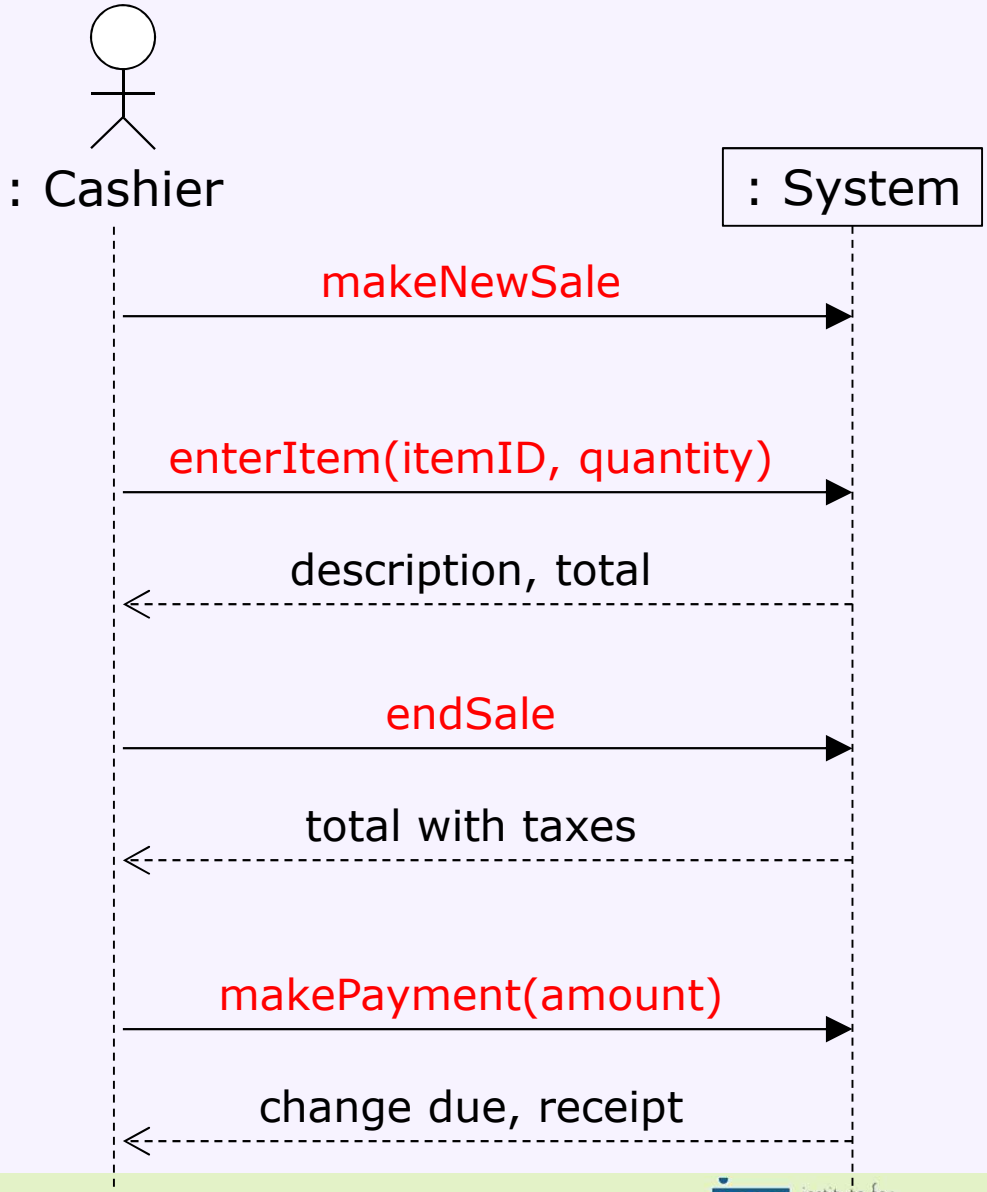
- Sequence diagrams are better to visualize the order in which things occur
- Communication diagrams also illustrate how objects are statically connected
- Communication diagrams often are more compact
- You should generally use interaction diagrams when you want to look at the behavior of several objects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

# Steps in the Design Process

- Precondition: understand functional requirements
  - Domain modeling
  - System sequence diagrams
  - **Behavioral contracts**
- Precondition: understand quality attribute requirements
- Design a logical architecture
- Design a behavioral model
  - (Sub)system sequence diagrams
  - **Behavioral contracts**
- Responsibility assignment
- Interface design
- Algorithm and data structure design – pseudo-code

# Behavioral Contracts: What do These Operations Do?

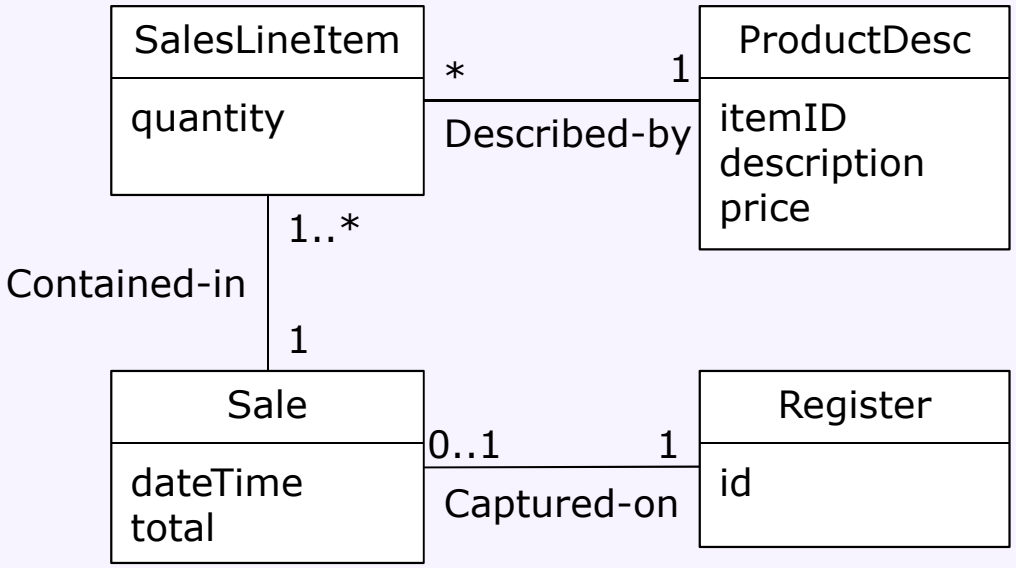
- To design, we need a spec
  - Preconditions
  - Postconditions
- We can write a *behavioral contract*
  - Like a pre-/post-condition specification for code
  - Often written in natural language
  - Focused on system interfaces
    - may or may not be methods



# 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 *(not discussed in 15-215)*
  - Preconditions
  - Postconditions

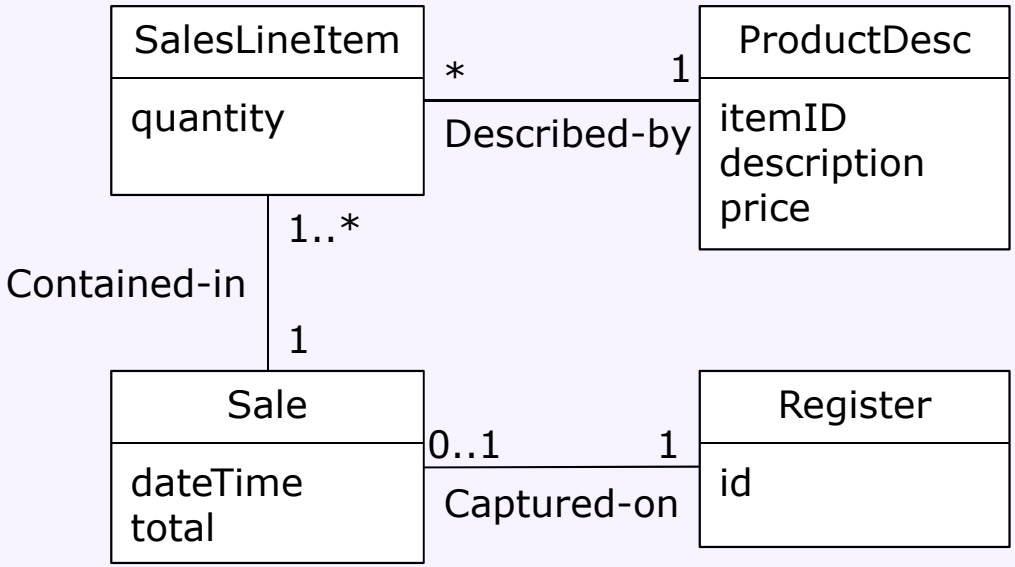
- Which contracts to write?
  - Operations that are complex or subtle
  - Operations that not everyone understands
  - Simple/obvious operations are often not given contracts in practice

- 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:**

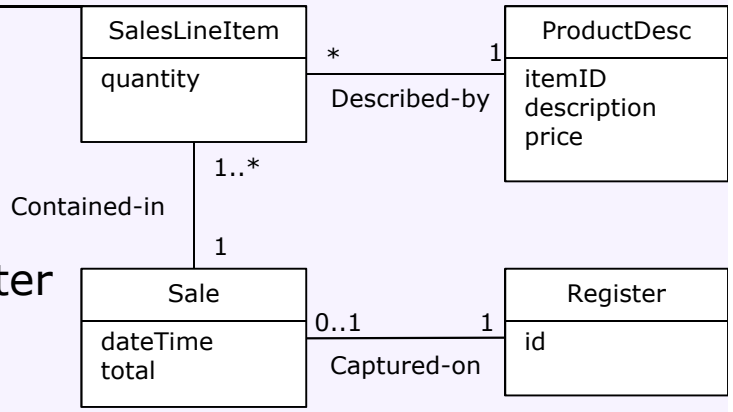
**Postconditions:**



# Example Point of Sale Contracts

**Operation:** makeNewSale()
**Preconditions:** none

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



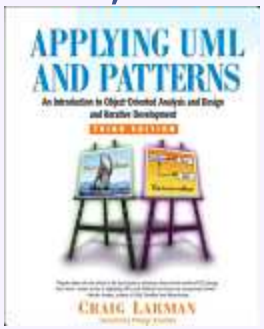

**Operation:** enterItem(itemID : ItemID, quantity : integer)
**Preconditions:** There is a sale s underway

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

# Literature on OO Design

- Alan Shalloway and James Trott. Design Patterns Explained, Addison Wesley, 2004
  - Brief introduction to UML
  - Introduction to design with design patterns
  - Mandatory reading


- Craig Larman, Applying UML and Patterns, Prentice Hall, 2004
  - Introduction to UML
  - Excellent discussion of object-oriented analysis and object-oriented design with and without patterns
  - Detailed additional material, many guidelines




- Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997
  - Detailed discussion of design goals and modularity

# Toad's Take-Home Messages

- Domain Modeling is a useful way to build understanding of the target domain
- Domain classes often turn into Java classes
  - However, sometimes they don't need to be modeled in code
  - Furthermore, some concepts only live at the code level
- UML is a commonly understood notation for domain modeling
- System Sequence Diagrams describe the overall operation of a (sub)system
- Behavioral contracts specify what program operations should do

# Bonus: Code Design Principles to Live By

- Or at least to do Homework 1 by!

- Don't repeat yourself
  - Avoid duplicate code—instead create a method and call it twice

- Separate concerns
  - Isolate each issue to as small a piece of code as you can
    - Within a single method
    - Within a subset of methods in a class
    - Within a single class
    - Within a single package
  - Each piece of code should be as *cohesive* as possible
  - Each piece of code should be *coupled* to other code as little as possible

# ArraySet – is this Good Code?

the special **length** field can be used to get the array size

```
class ArraySet {
    private int members[]; // the array is sorted
    private int temp[]; // for performing unions
    public ArraySet union (ArraySet s) {
        if (temp == null || size + s.size > temp.length)
            temp = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
    // other methods not shown
}
```

# ArraySet – is this Good Code?

the special **length** field can be used to get the array size

```
class ArraySet {
    private int members[]; // the array is sorted
    private int temp[]; // for performing unions
    public ArraySet union (ArraySet s) {
        if (temp == null || size + s.size > temp.length)
            temp = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
    // other methods not shown
}
```

***coupling*** **means one part of the code depends on another part. Some coupling is necessary, but unnecessary coupling makes code harder to understand and modify.**

**This code has high coupling because the details of the union algorithm depend on a field in `ArraySet` that is used for no other purpose. Making `temp` a local variable would reduce coupling.**