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

Object-oriented analysis: Modeling a problem domain

Christian Kästner Bogdan Vasilescu

School of Computer Science

institute for SOFTWARE RESEARCH

LEFTOVER TOPICS FROM DESIGN FOR REUSE

Generics / Parametric Polymorphism

```
Map<Integer,String> …;
public class Stack<T> {
        public void push(T obj) { … }
        public T pop() { … }
}
abstract class Foo {
        abstract public <T> T process(List<T> l);
}
ArrayList<? extends Animal> … ;
```

The Iterator Design Pattern

- Generic traversal of elements of a data structure or computation
  - All items in a list, set, tree
  - The Fibonacci numbers
  - All permutations of a set
- Interface:
  - hasNext(), next()
  - example: **while** (i.hasNext()) { x = i.next(); process(x); } *or* **for** (x : i) { process(x); }

Iterators in Java

```
interface Iterable<E> {//implemented by most collections
        Iterator<E> iterator();
}
interface Iterator<E> {
        boolean hasNext();
        E next();
        void remove();    // removes previous returned item
}                         // from the underlying collection
```

An Iterator implementation for Pairs

```
public class Pair<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second=s;}
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

An Iterator implementation for Pairs

```
public class Pair<E> implements Iterable<E> {
  private final E first, second;
  public Pair(E f, E s) { first = f; second=s;}
  public Iterator<E> iterator() {
    return new PairIterator();
  }
  private class PairIterator implements Iterator<E> {
    private boolean seenFirst = false, seenSecond = false;
    public  boolean hasNext() { return !seenSecond; }
    public  E next() {
      if (!seenFirst)  { seenFirst  = true; return first;  }
      if (!seenSecond) { seenSecond = true; return second; }
      throw new NoSuchElementException();
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }
}
```

```
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { … }
```

Fibonacci Iterator

```
class FibIterator implements Iterator<Integer> {
        public  boolean hasNext() { return true; }
        public  Integer next() {




        }
        public void remove() {
          throw new UnsupportedOperationException();
        }
}
```

Fibonacci Iterator

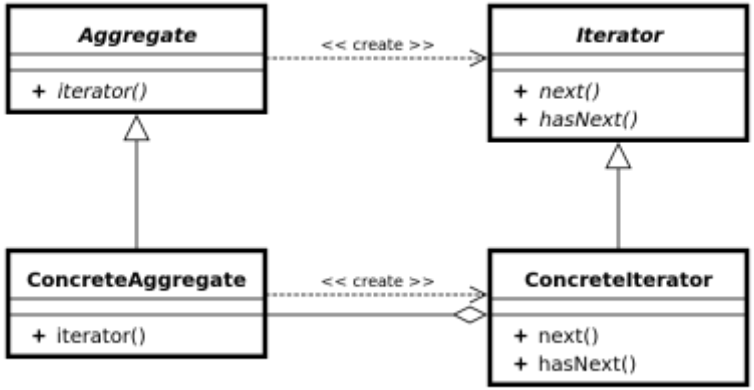
```
class FibIterator implements Iterator<Integer> {
        public  boolean hasNext() { return true; }
        private int a = 1;
        private int b =1;
        public  Integer next() {
                int result = a;
                a = b;
                b = a + result;
                return result;
        }
        public void remove() {
          throw new UnsupportedOperationException();
        }
}
```

Using a java.util.Iterator<E>: A warning

- The default Collections implementations are mutable…
- …but their Iterator implementations assume the collection does not change while the Iterator is being used
  - You will get a ConcurrentModificationException
  - If you simply want to remove an item:

```
List<String> arguments = …;
for (Iterator<String> it = arguments.iterator();
     it.hasNext();  ) {
  String s = it.next();
  if (s.equals("Charlie"))
    arguments.remove("Charlie"); // runtime error
                  // use it.remove instead
}
```

The Iterator Pattern



- Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  - Order is unspecified, but access every element once
- Solution: A strategy pattern for iteration
- Consequences:
  - Hides internal implementation of underlying container
  - Easy to change container type
  - Facilitates communication between parts of the program

Summary: Design for Reuse

- Delegation and inheritance to reuse code
  - implements vs extends, abstract classes
  - Strategy vs template method pattern
  - Flexible delegation: Decorator design pattern
- Invariants and constructors
- Contracts and inhertiance
  - Behavioral subtyping/Liskov substitution principle
- Immutable objects
- Parametric polymorphism (generics)

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

Object-oriented analysis: Modeling a problem domain

Christian Kästner Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Learning Goals

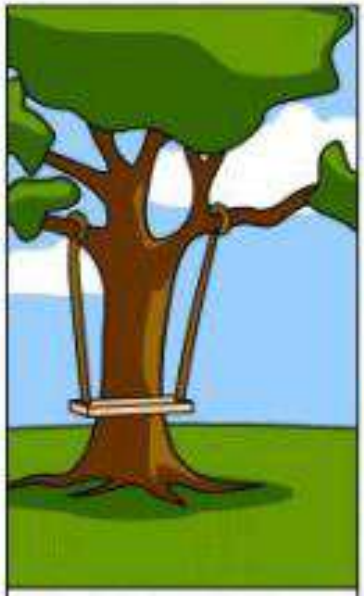
- High-level understanding of requirements challenges
- Identify the key abstractions in a domain, model them as a domain model
- Identify the key interactions within a system, model them as system sequence diagram
- Discuss benefits and limitations of the design principle low representational gap

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

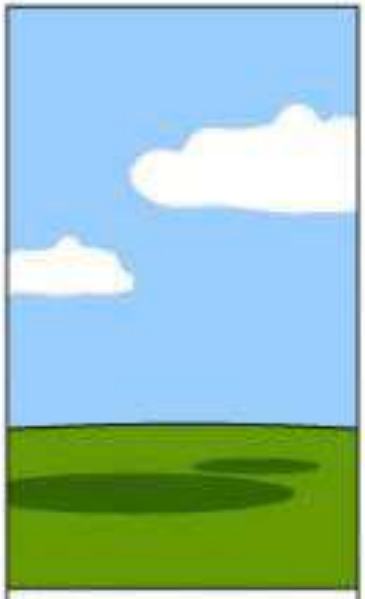


How the Project Leader understood it

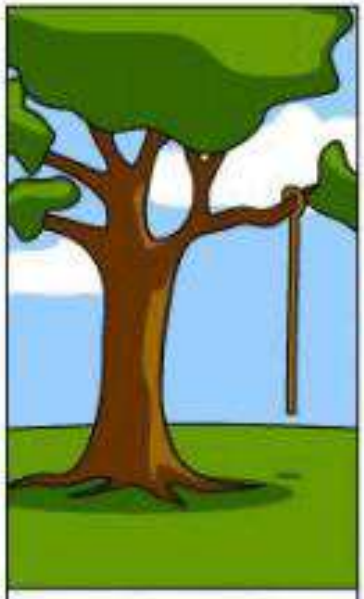
How the Analyst designed it

How the Programmer wrote it

How the Business Consultant described it

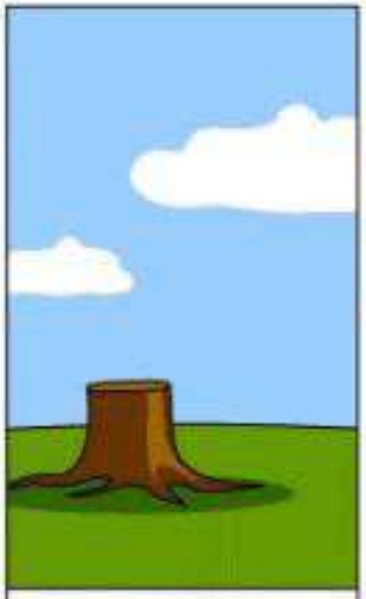


How the project was documented



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 the rotation gain and condition 2

Requirements

- Wh
- Wh
  Leg
- Cus ally want; vague, biased by what they see; change th
- D
- (A
  building the thing right?)

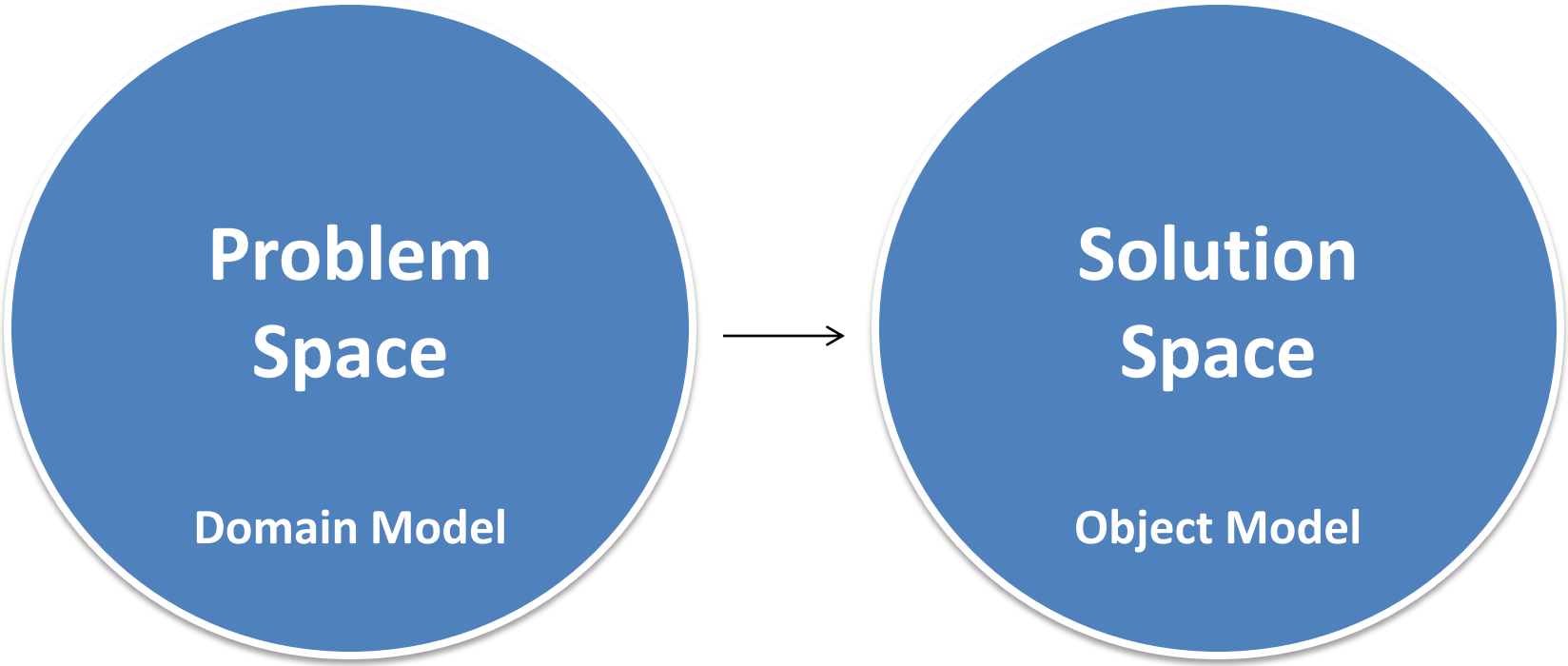
214 assumption:
Somebody has gathered the requirements (mostly text).

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

Human and social issue
15-313 topic

This lecture

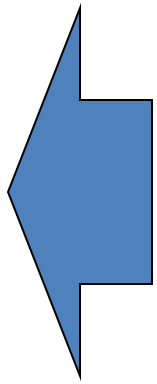
- Understand functional requirements
- Understand the problem's vocabulary (domain model)
- Understand the intended behavior (system sequence diagrams; contracts)



- 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

A high-level software design process

- Project inception
- Gather requirements
- Define actors, and use cases
- Model / diagram the problem, define objects
- Define system behaviors
- Assign object responsibilities
- Define object interactions
- Model / diagram a potential solution
- Implement and test the solution
- Maintenance, evolution, …

15-313

15-214

…

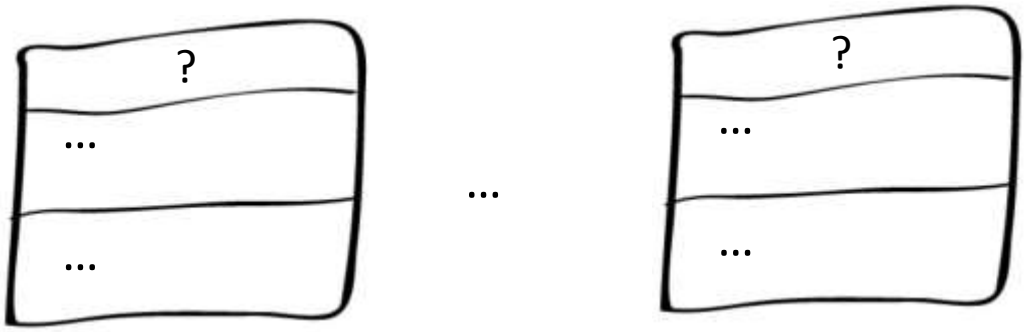
DESIGN PRINCIPLE:
LOW REPRESENTATIONAL GAP

Representational gap

- Real-world concepts:



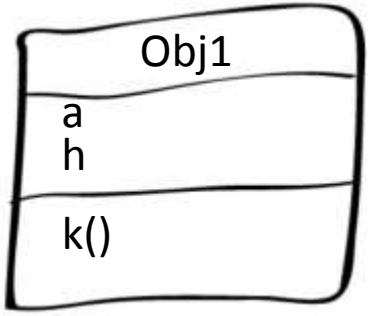
- Software concepts:

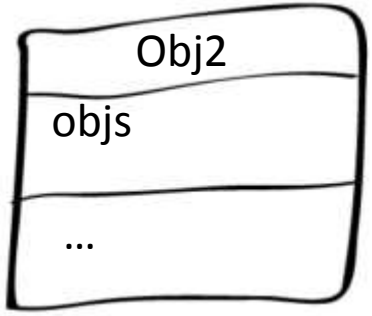


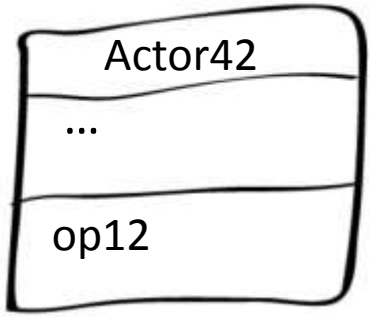
Representational gap

- Real-world concepts:

- Software concepts:



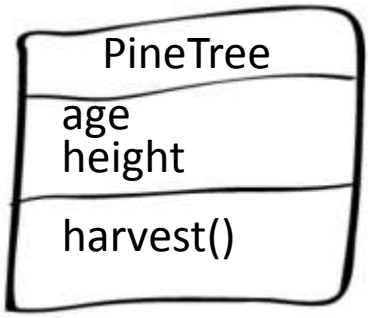


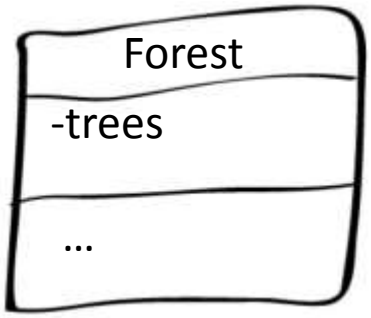


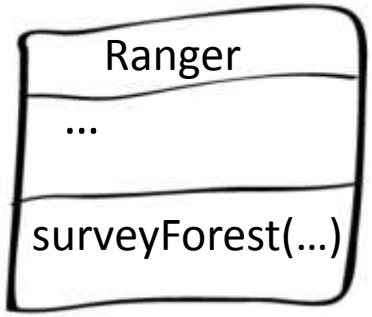
Representational gap

- Real-world concepts:

- Software concepts:





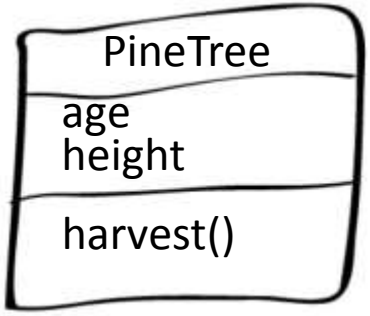


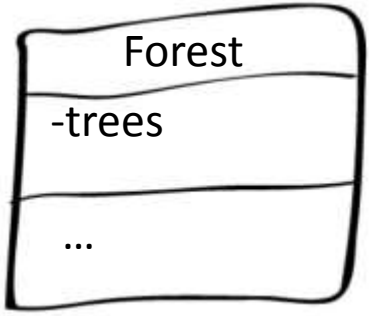
Representational gap

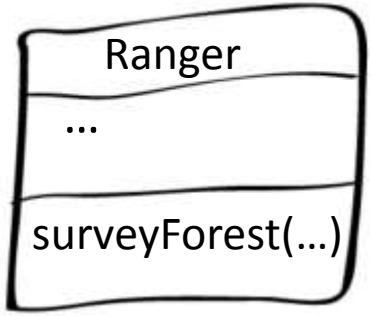
- Real-world concepts:

inspires objects and names

- Software concepts:







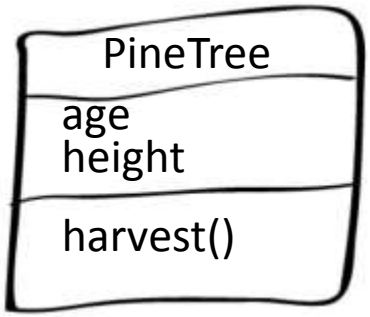
Representational gap

- Real-world concepts:

Problem Space

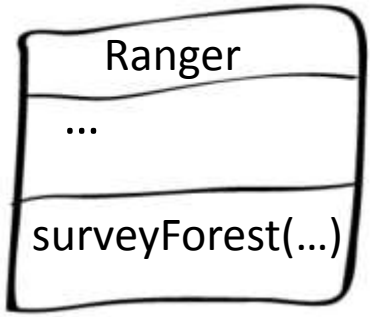
Domain Model

- Software concepts:



Solution Space

Object Model



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

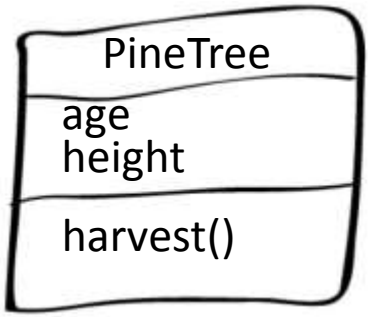
=> Find and understand real-world objects and abstractions

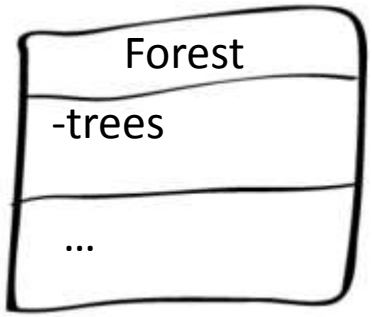
Benefit of Low Representational Gap (Congruence)

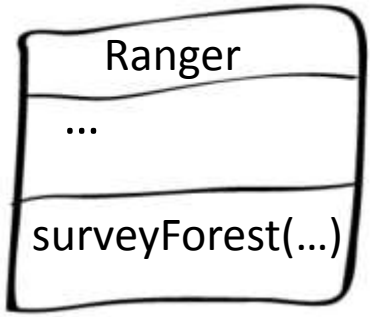
- The domain model is familiar to domain experts
  - Simpler than code
  - Uses familiar names, relationships
- Classes in the object model and implementation will be inspired by domain model
  - similar names
  - possibly similar connections and responsibilities
- Facilitates understanding of design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

A related design principle: high cohesion

- Each component should have a small set of closely-related responsibilities
- Benefits:
  - Facilitates understandability
  - Facilitates reuse
  - Eases maintenance







DOMAIN MODELS

Artifacts of this design process

- Model / diagram the problem, define objects
  - Domain model (a.k.a. conceptual model)
- Define system behaviors
  - System sequence diagram
  - System behavioral contracts
- Assign object responsibilities, define interactions
  - Object interaction diagrams
- Model / diagram a potential solution
  - Object model

Today: understanding the problem

Defining a solution

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

Input to the design process: Requirements and use cases

- Typically prose:

> 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, …

Running Example in Book

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

Modeling a problem domain

- Identify key concepts of the domain description
  - Identify nouns, verbs, and relationships between concepts
  - Avoid non-specific vocabulary, e.g. "system"
  - Distinguish operations and concepts
  - Brainstorm with a domain expert
- Visualize as a UML class diagram, a domain model
  - Show class and attribute concepts
    - Real-world concepts only
    - No operations/methods
    - Distinguish class concepts from attribute concepts
  - Show relationships and cardinalities

Building a domain model for a library system

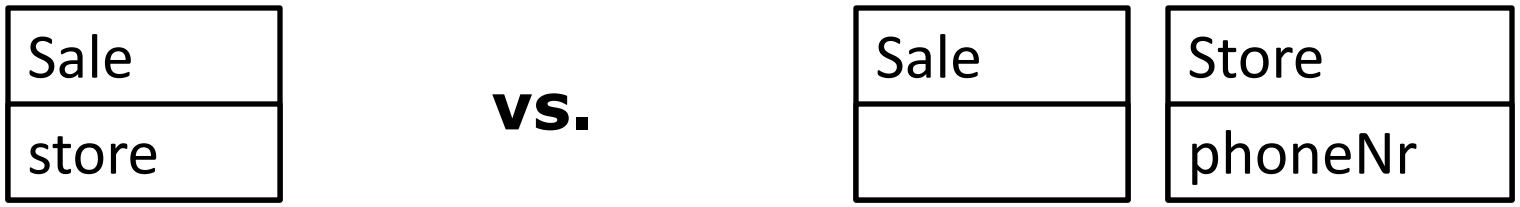
- A public library typically stores a collection of books, movies, or other library items available to be borrowed by people living in a community. Each library member typically has a library account and a library card with the account's ID number, which she can use to identify herself to the library.
- A member's library account records which items the member has borrowed and the due date for each borrowed item. Each type of item has a default rental period, which determines the item's due date when the item is borrowed. If a member returns an item after the item's due date, the member owes a late fee specific for that item, an amount of money recorded in the member's library account.

One domain model for the library system

Notes on the library domain model

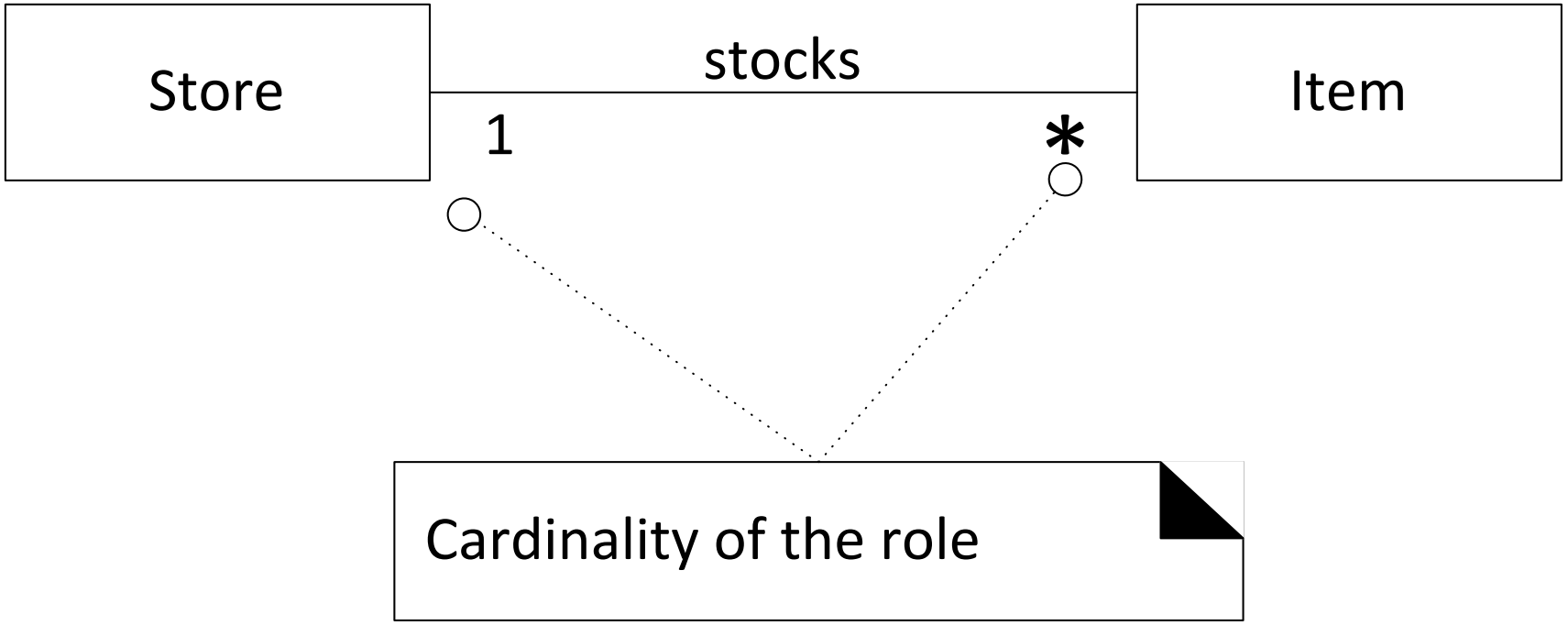
- All concepts are accessible to a non-programmer
- The UML is somewhat informal
  - Relationships are often described with words
- Real-world "is-a" relationships are appropriate for a domain model
- Real-word abstractions are appropriate for a domain model
- Iteration is important
  - This example is a first draft. Some terms (e.g. Item vs. LibraryItem, Account vs. LibraryAccount) would likely be revised in a real design.
- Aggregate types are usually modeled as classes
- Primitive types (numbers, strings) are usually modeled as attributes

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

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 (see textbook for details)

- 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

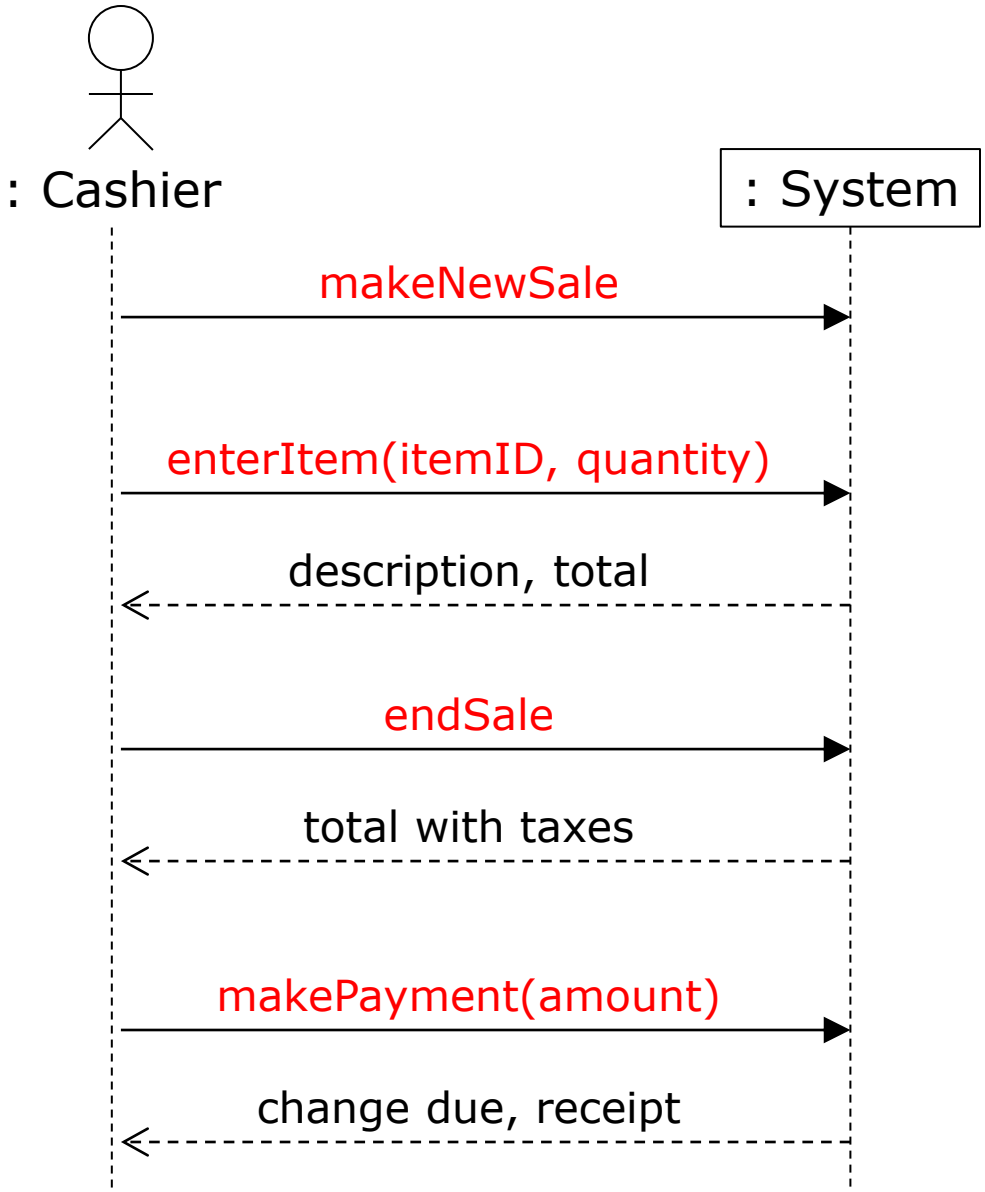
Three perspectives of class diagrams

- 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
  - Software classes representing candidates for implem.
  - Often confused in OO since classes combine both interfaces and implementation
- Implementation: Diagram describes actual implementation classes

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


: Cashier
: System
makeNewSale
enterItem(itemID, quantity)
description, total
endSale
total with taxes
makePayment(amount)
change due, receipt

System Sequence Diagrams

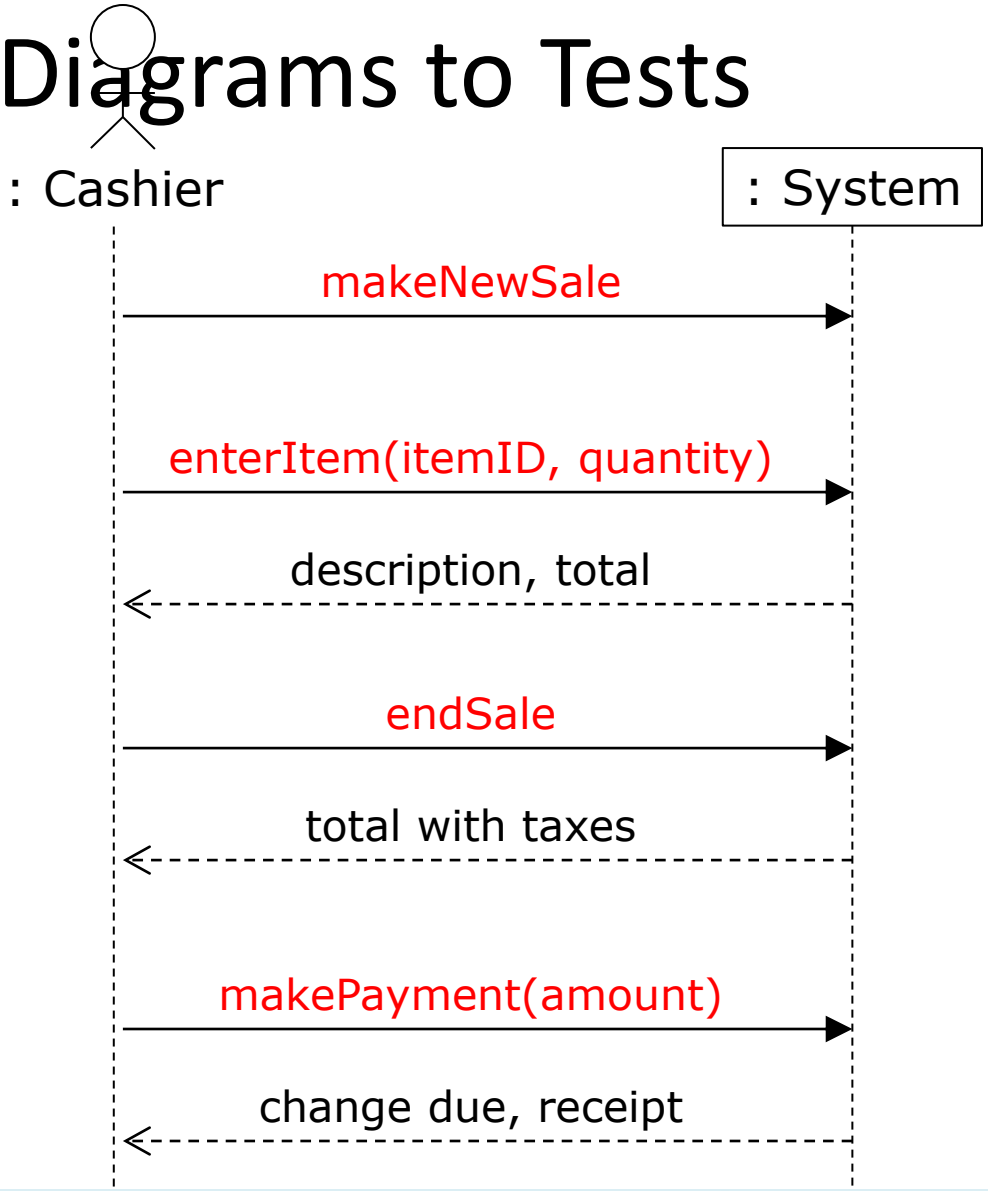
- A *system sequence diagram* is a model that shows, for one scenario of use, the sequence of events that occur on the system's boundary
- Design goal: **Identify and define the interface of the system**
  - Two components: A user and the overall system
  - Useful for identifying tests later
- Input: Domain description and one use case
- Output: A sequence diagram of system-level operations
  - Include only domain-level concepts and operations

One sequence diagram for the library system

Use case scenario: A library member should be able to use her library card to log in at a library system kiosk and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its rental period to the current day, and record the book and its due date as a borrowed item in the member's library account.

Outlook:
System Sequence Diagrams to Tests

```
s = new System();
a = s.makeNewSale();
t = a.enterItem(…);
assert(50.30, t);
tt = a.endSale();
assert(52.32, tt);
…
```



Interaction diagrams

- See textbook for notation of UML communication and sequence diagrams

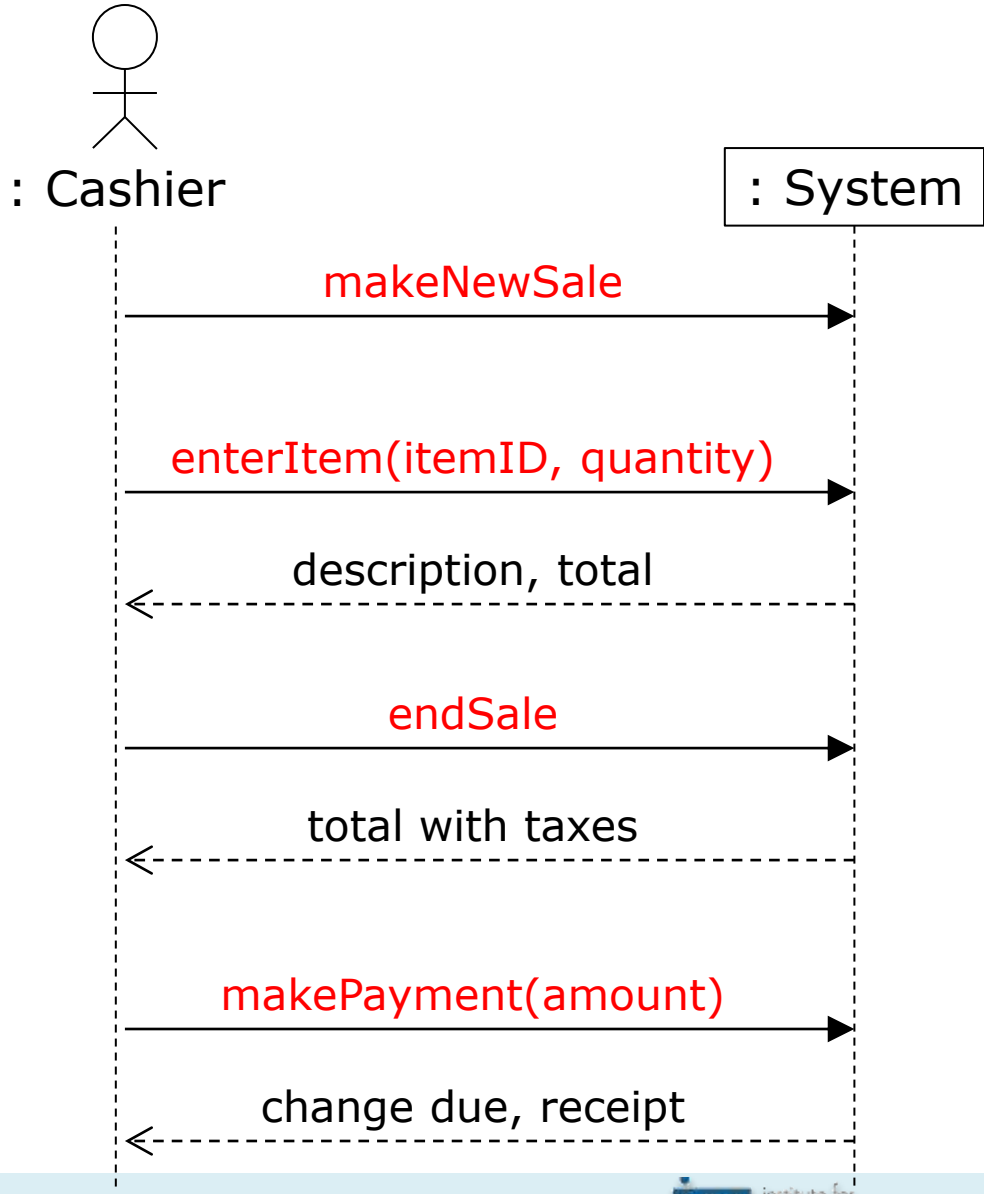
Sequence vs Communication Diagrams

- 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.

SYSTEM BEHAVIORAL CONTRACTS

Behavioral Contracts: What do These Operations Do?

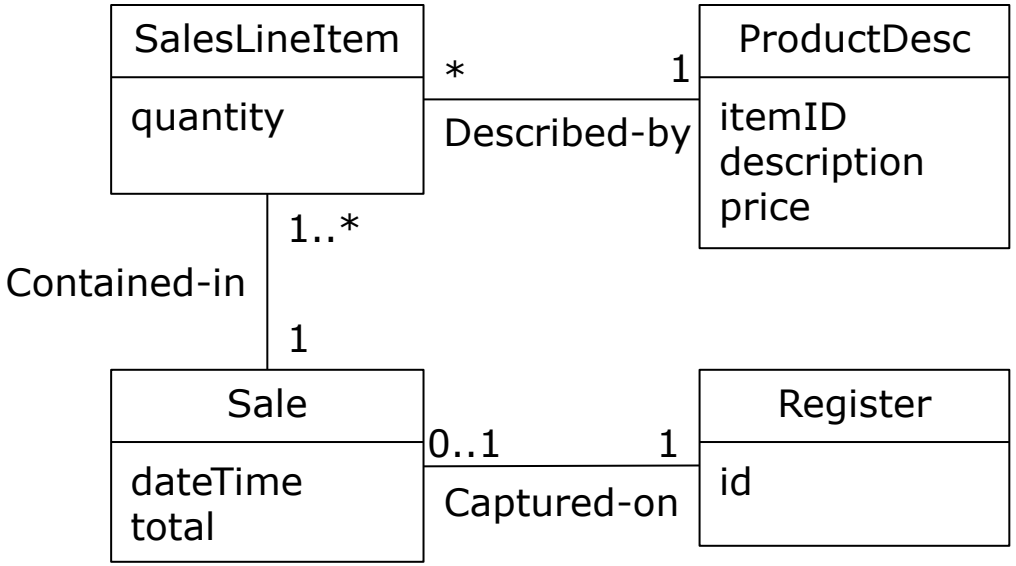
- Behavioral contract
  - Like a pre-/post-condition specification for code
  - Often written in natural language
  - **Focused on system interfaces**
    - may or may not be methods



Example Point of Sale Contract

Operation: makeNewSale()
Preconditions: none

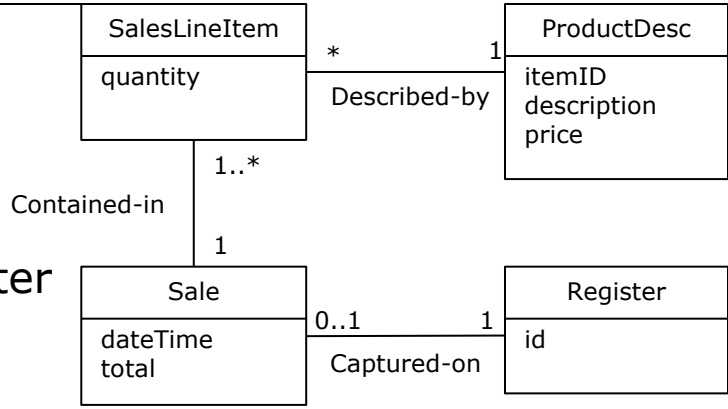
Postconditions: - A Sale instance s was created
- s was associated with a Register



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

A system behavioral contract for the library system

- Operation: borrow(item)
- Pre-conditions:

- Post-conditions:

Distinguishing domain vs. implementation concepts

- Domain-level concepts:
  - Almost anything with a real-world analogue
- Implementation-level concepts:
  - Implementation-like method names
  - Programming types
  - Visibility modifiers
  - Helper methods or classes
  - Artifacts of design patterns

Take-Home Messages

- To design a solution, problem needs to be understood
- Know your tools to build domain-level representations
  - Domain models – understand domain and vocabulary
  - System sequence diagrams + behavioral contracts – understand interactions with environment
- Be fast and (sometimes) loose
  - Elide obvious(?) details
  - Iterate, iterate, iterate, …
- 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
- Get feedback from domain experts
  - Use only domain-level concepts