Principles of Software Construction: Objects, Design, and Concurrency

Object-oriented analysis

Christian Kästner Vincent Hellendoorn

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrativa

Recitations C and E overloaded, please consider alternatives if not registered for this one

- Next week's recitation is not language-specific

hw3 to be released soon (modeling + coding), covers material from today, Tuesday, readings, and Wednesday

Expect hw1 grades mid next week

Some Testing Hints

Code may be used in many contexts, don't make assumptions based on one client

Code only pushes values larger than prior ones in this implementation.

Is this true for all users of Queue?

```
q = new Queue();
last = 0;
for (...) {
    value = read();
    if (value > last)
        q.push(value);
}
```

Some Testing Hints

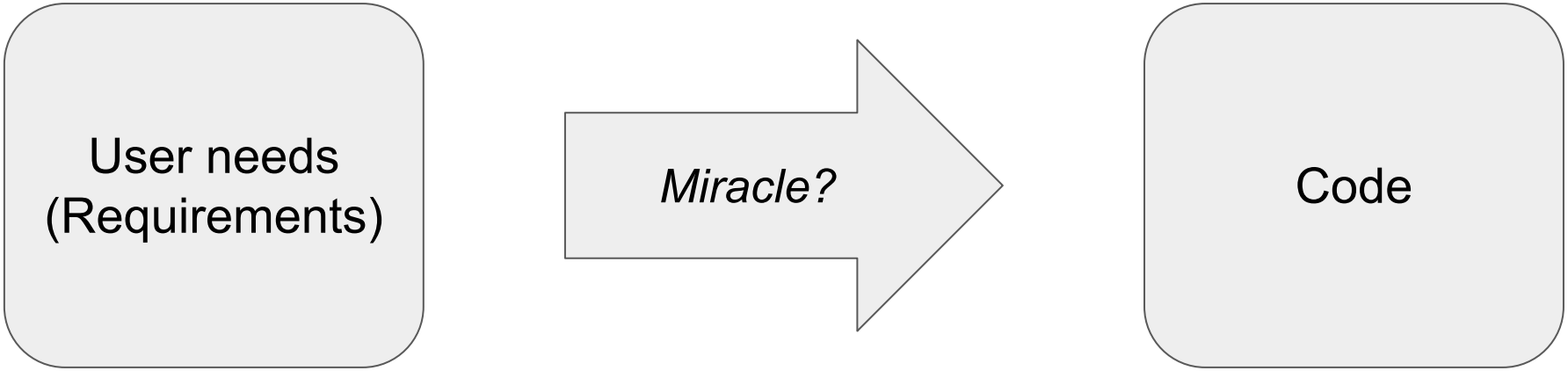
Testing code with dependencies

```
@Test ...
Comparator x =
   myComplexImpl();
List l =
   loadFromFile();
l.sort(x);
```

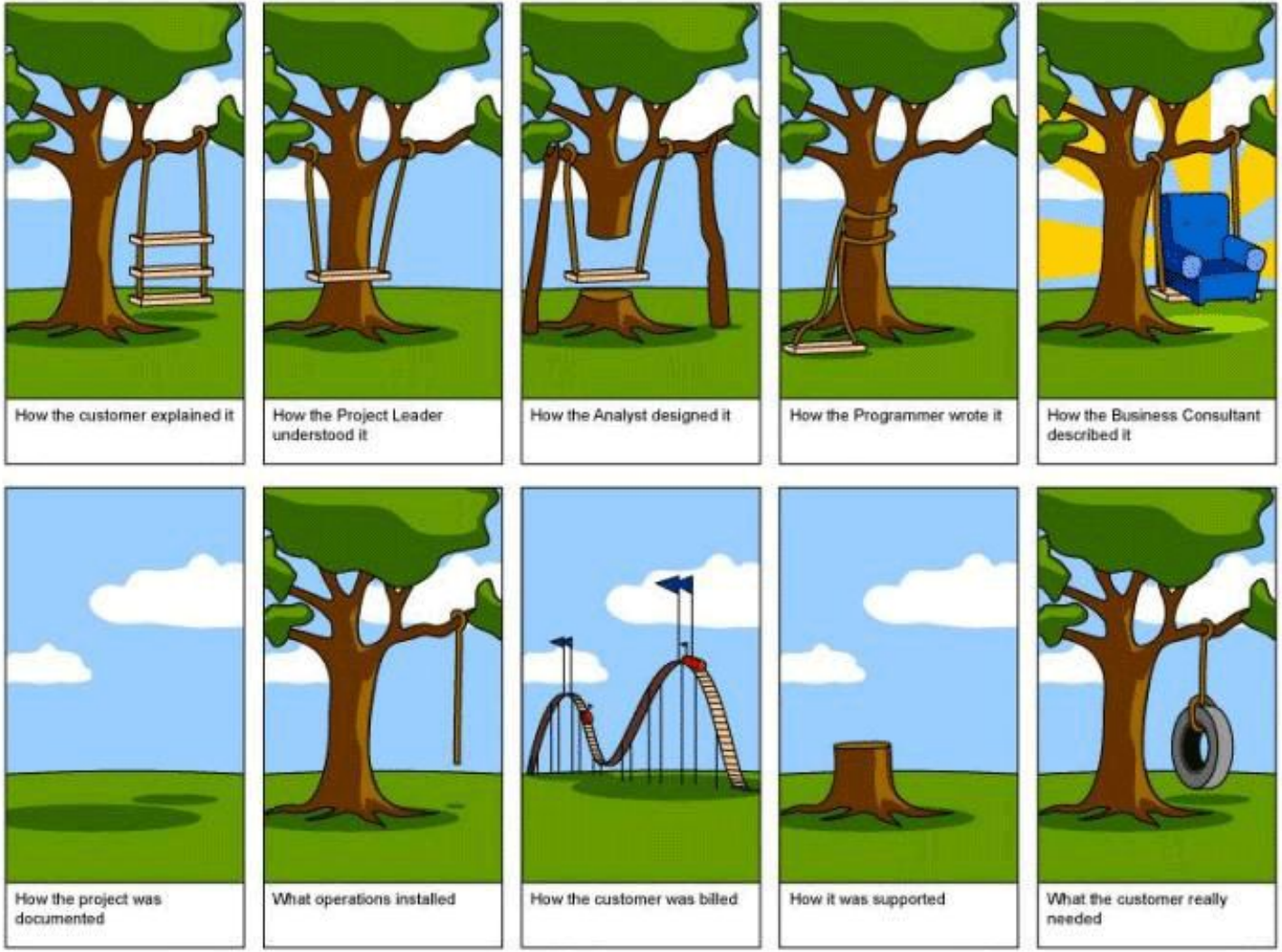
If testing *sort*, avoid unnecessary dependencies. Simple implementations of other objects sufficient.

Learning Goals

- High-level understanding of requirements challenges
- Use basic UML notation to communicate designs
- 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


User needs
(Requirements)
Miracle?
Code

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 beyond our scope (see 17-313)

Lufthansa Flight 2904

- The Airbus A320-200 airplane has a software-based braking system
- Engaging reverse thrusters while in the air is very dangerous: **Only allow breaking when on the ground**

Lufthansa Flight 2904

Two conditions needed to “be on the ground”:

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

Requirements

- What
- What
- Custo
- Diffic
- (Are
right?

Assumption in this course: Somebody has gathered most requirements (mostly text).

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

beyond our scope (see 17-313)

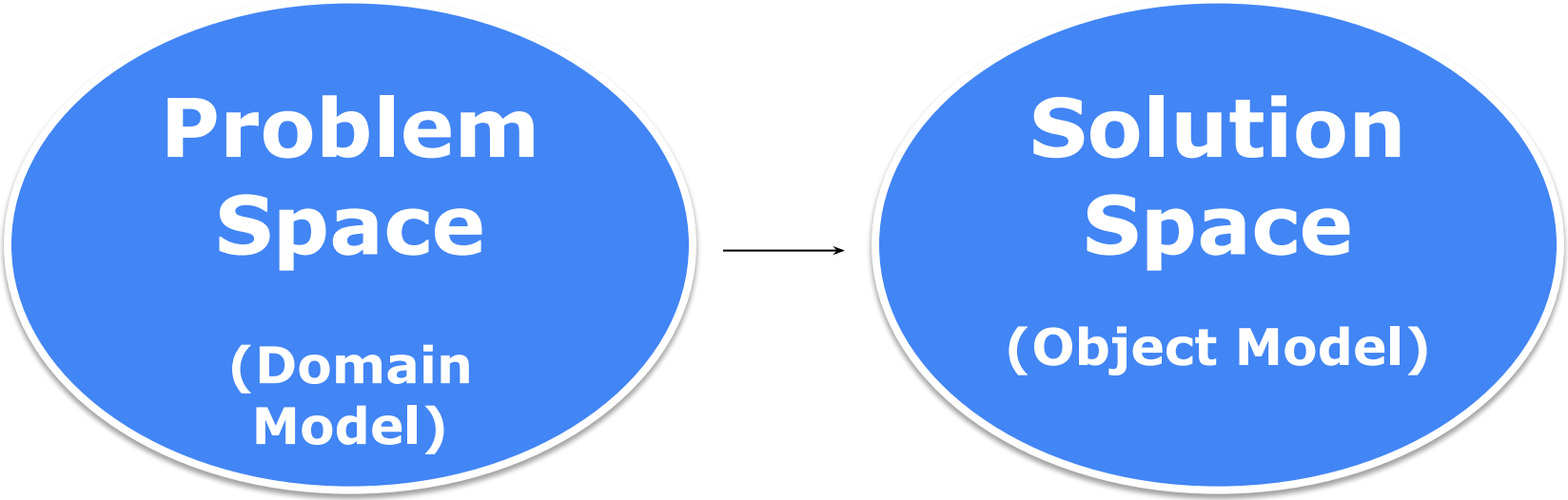
This lecture

Understand functional requirements

Understand the problem's vocabulary (domain model)

Understand the intended behavior (system sequence diagrams; contracts)

UML as a design language



- 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

An object-oriented design process

Model / diagram the problem, define concepts

- **Domain model** (a.k.a. conceptual model), **glossary**

Define system behaviors

- **System sequence diagram**
- **System behavioral contracts**

Assign object responsibilities, define interactions

- **Object interaction diagrams**

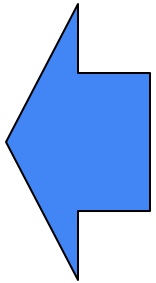
Model / diagram a potential solution

- **Object model**

OO Analysis: Understanding the problem

OO Design: Defining 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

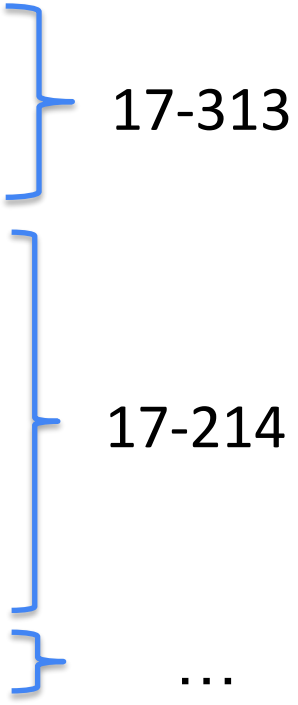
- Identify software classes and their relationships with class diagrams
- Assign responsibilities (attributes, methods)
- Explore behavior with interaction diagrams
- Explore design alternatives
- Create an object model (aka design model) and interaction models

Implementation

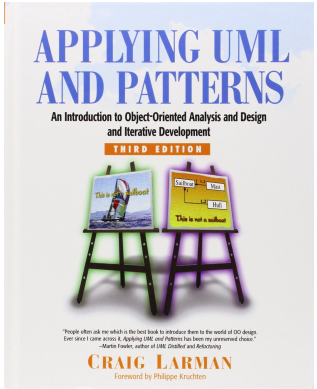
- Map designs to code, implementing classes and methods

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, …



DOMAIN MODELS



Chapter 9

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)

Input to the analysis process: Requirements and use cases

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 must pay a late fee
member's library account.

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.

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

Concepts in 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.

Glossary

Identify and define key concepts

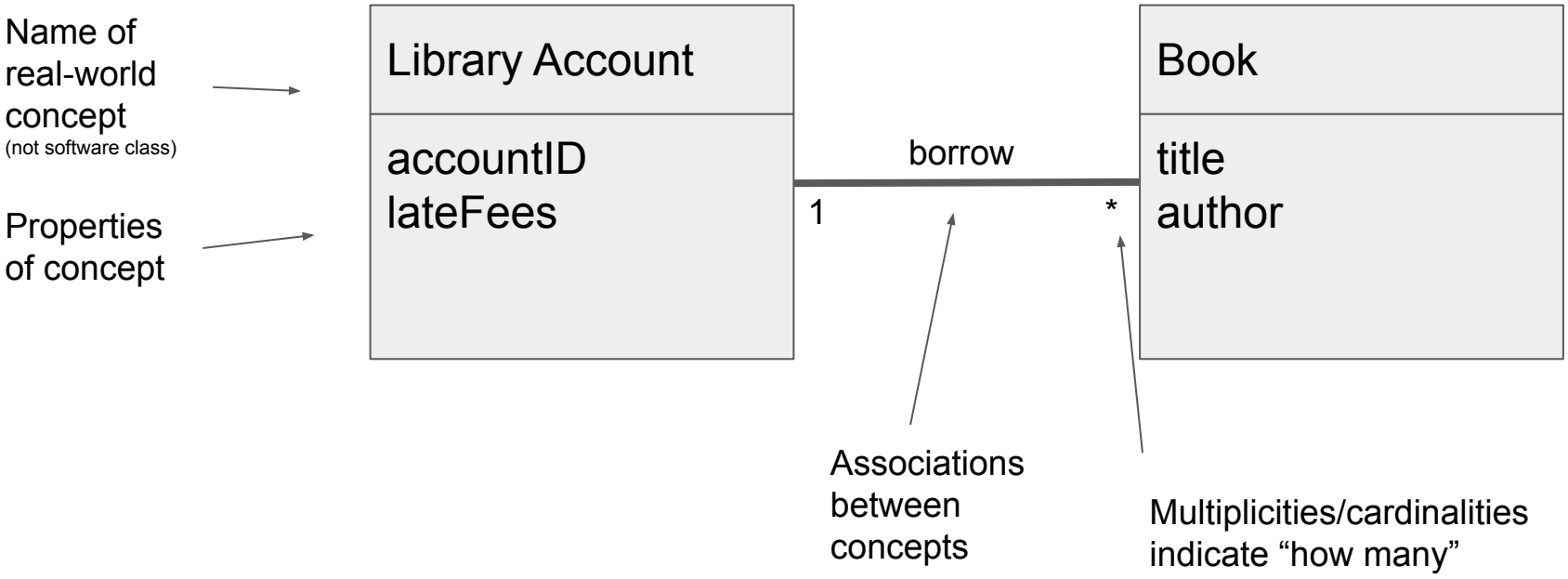
Ensure shared understanding between developers and customers

Library item: Any item that is indexed and can be borrowed from the library
Library member: Person who can borrow from a library, identified by a card with an ID number
Book

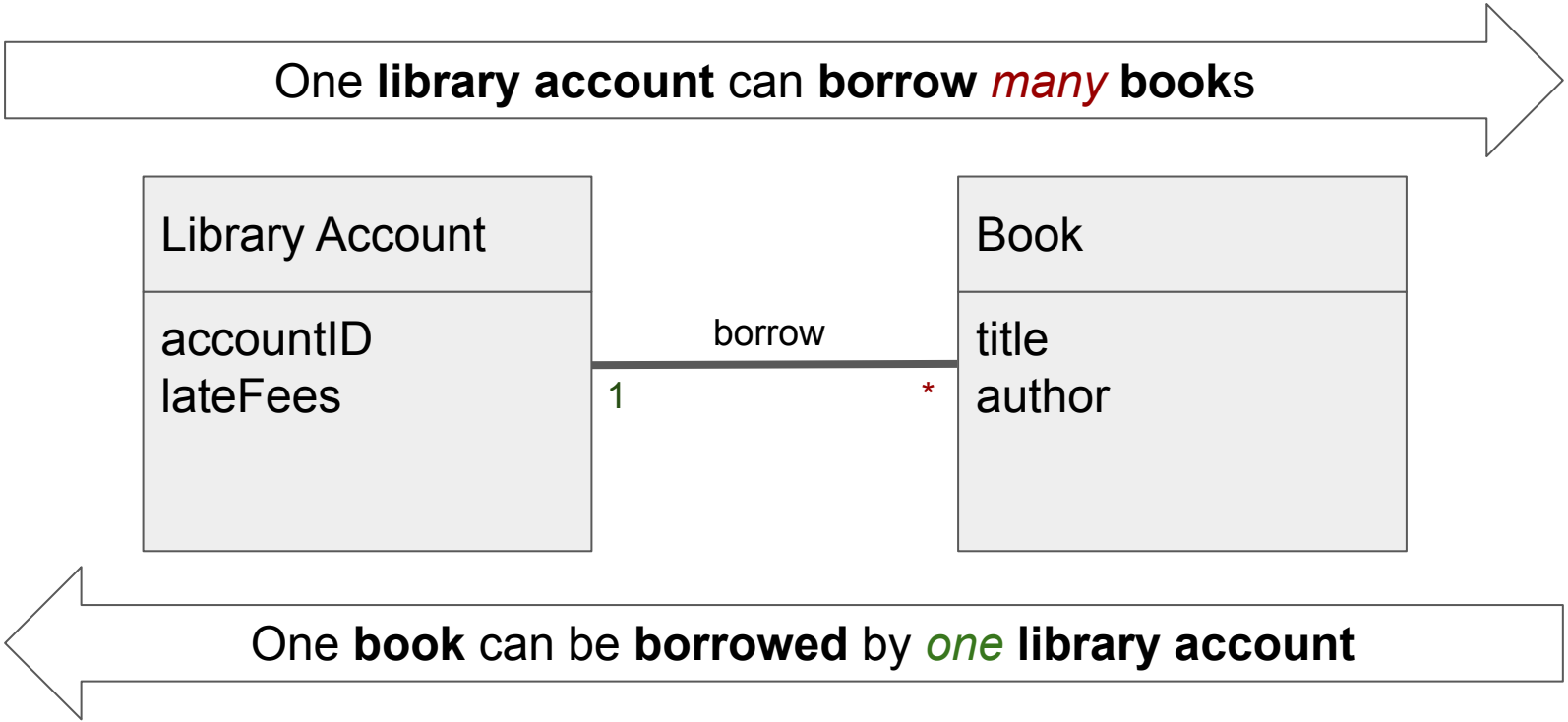
Define potentially ambiguous concepts

No need to expand on obvious concepts

Visual notation: UML

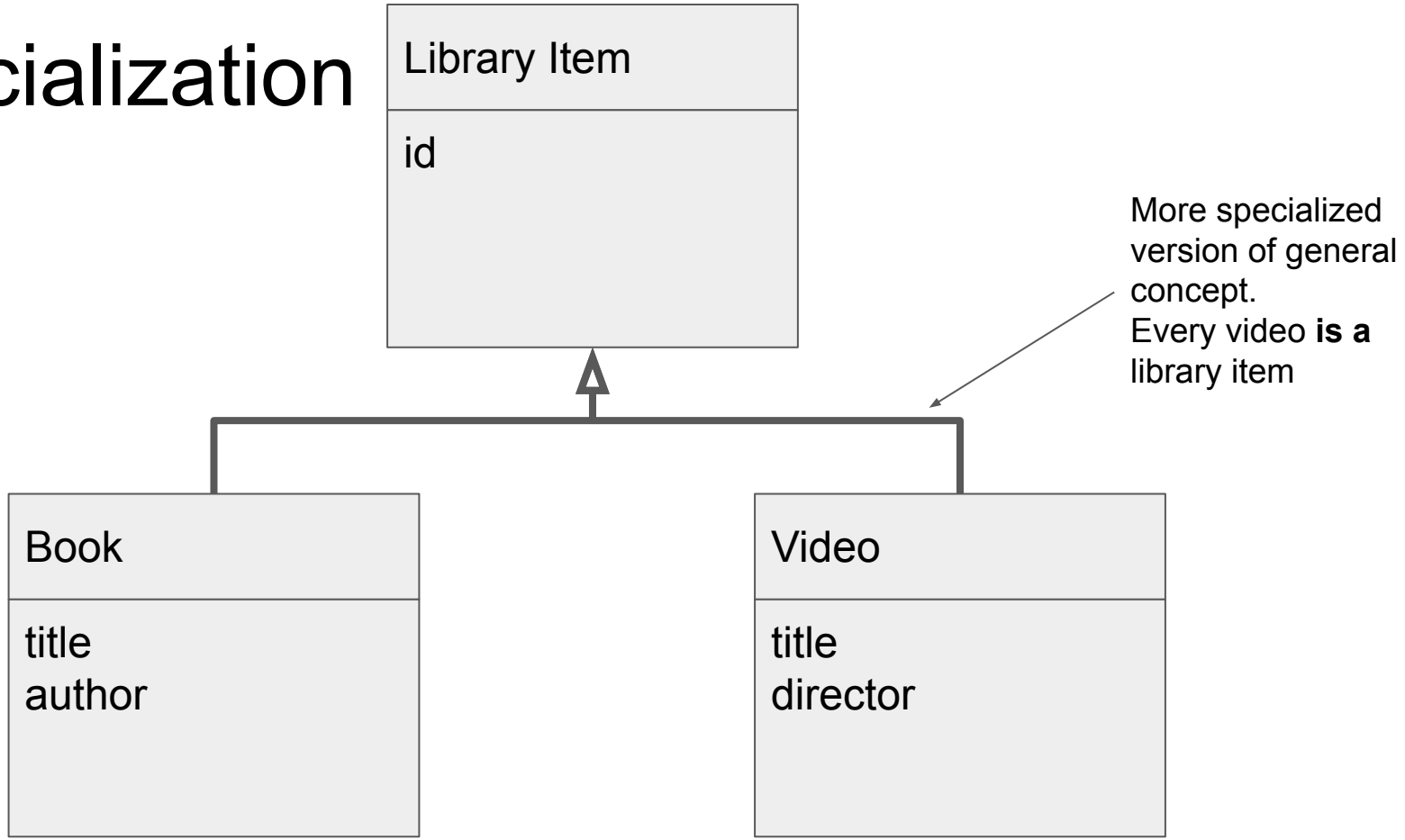


Reading associations



Reading associations

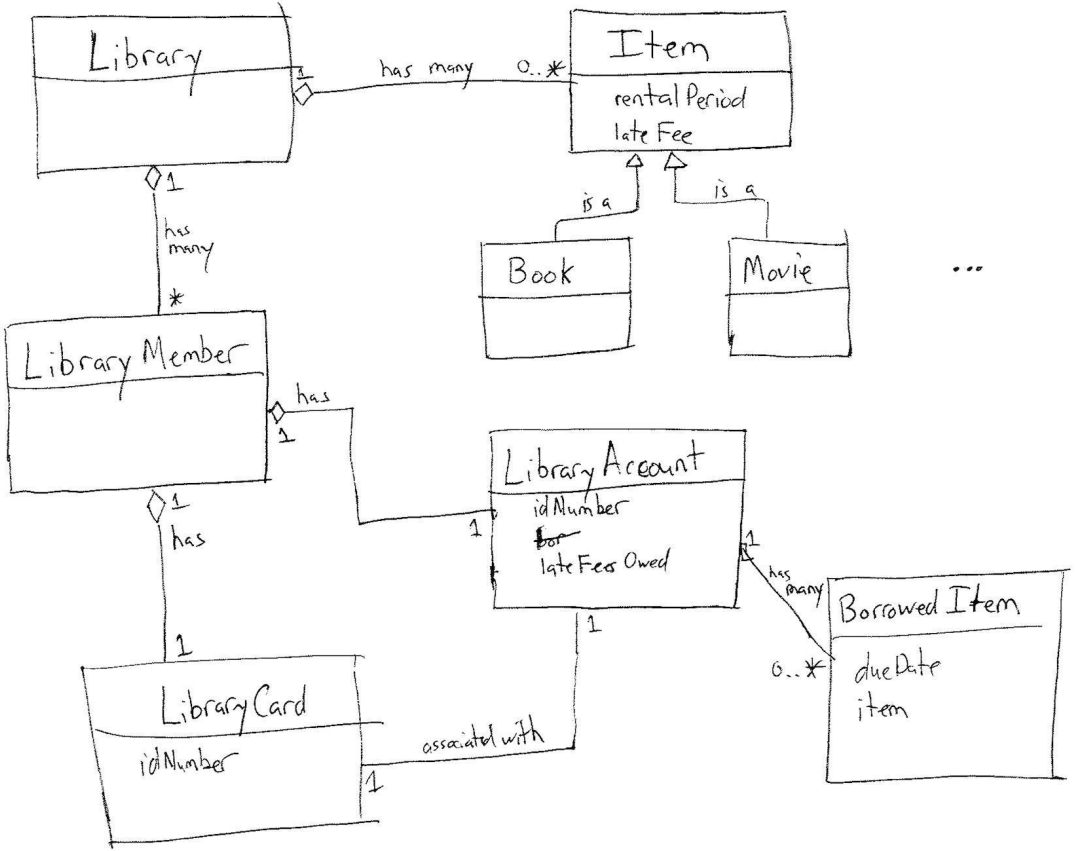
Specialization



Concepts 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 concept, not an attribute"
- Avoid type annotations

One domain model for the library system



Modeling a problem domain

A domain model is a living document

Used for communication

Focus on **real-world concepts**,

- Not abstract implementation concerns (e.g., database)
- No methods/operations
- Show relationships and cardinalities

Identifying concepts

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.

Identifying concepts

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.

Hints for Identifying Concepts

Read the requirements description, look for nouns

Reuse existing models

Use a category list

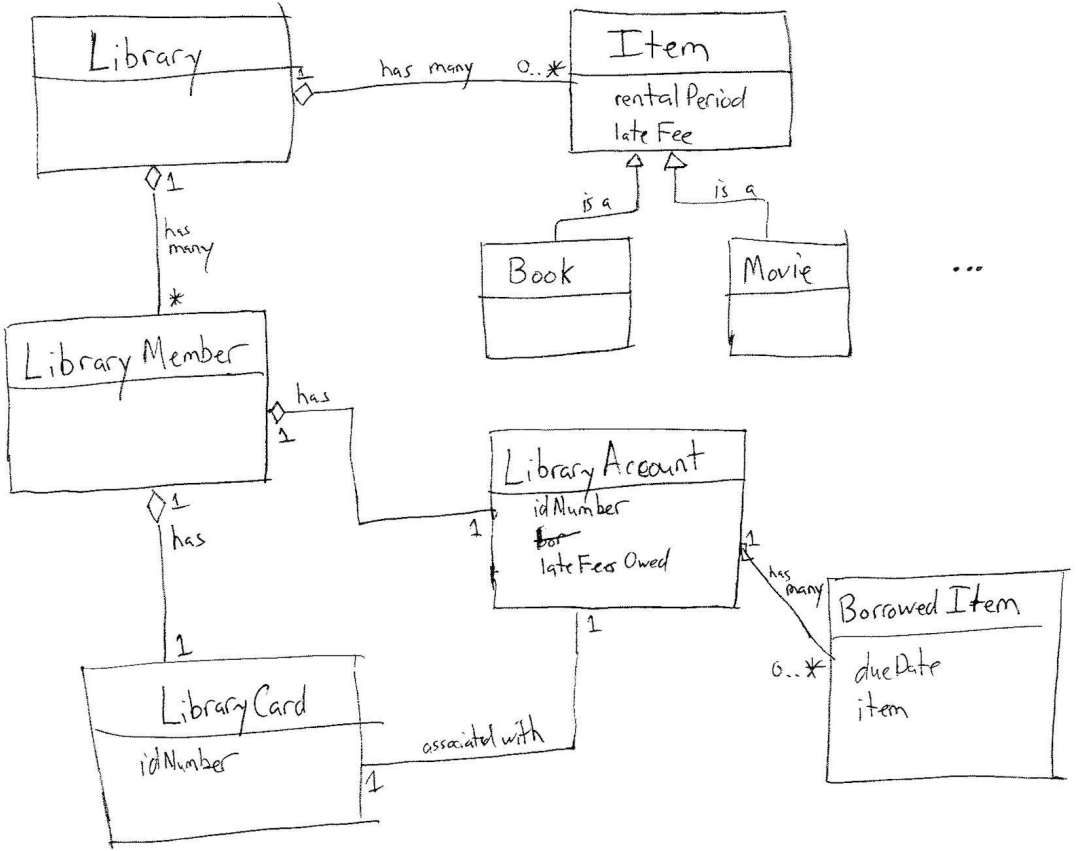
- tangible things: cars, telemetry data, terminals, …
- roles: mother, teacher, researcher
- events: landing, purchase, request
- interactions: loan, meeting, intersection, …
- structure, devices, organizational units, …

Analyze typical use scenarios, analyze behavior

Brainstorming

Collect first; organize, filter, and revise later

One domain model for the library system



Notes on the library domain model

- All concepts are accessible to a non-programmer
- UML notation somewhat informal; relationships 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 separate concepts
- Basic attributes (numbers, strings) are usually modeled as attributes

Why domain modeling?

Understand the domain

- Details matter! Are books different from videos for the system?

Ensure completeness

- Late fees considered?

Agree on a common set of terms

- library item vs collection entry vs book

Prepare to design

- Domain concepts are good candidates for OO classes (-> low representational gap)

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"

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.

Outlook: Build a domain model for Homework 3

Need Help?

Video Tutorials More of a visual learner? We've got you covered! Head over to roxley.com/santorini-video for video tutorials on how to play, as well as complete visual demonstrations of all God Powers!

Santorini App Can't decide which God Powers to match up? Head over to Google Play Store or the Apple App Store and download the Santorini App absolutely free. Complete with video tutorials, match randomizer and much more!

Setup

1. Place the smaller side of the Cliff Pedestal (A) on the Ocean Board (B), using the long and short tabs on the Cliff Pedestal to guide assembly.
2. Place the Island Board (C) on top of the Cliff Pedestal (A), again using the long and short tabs to guide assembly.
3. The youngest player is the Start Player, who begins by placing 2 Workers (D) of their chosen color into any unoccupied spaces on the board. The other player(s) then places their Workers (E).

How To Play

Players take turns, starting with the Start Player, who first placed their Workers. On your turn, select one of your Workers. You must move and then build with the selected Worker.

Move your selected Worker into one of the (up to) eight neighboring spaces .

A Worker may move up a maximum of one level higher, move down any number of levels lower, or move along the same level. A Worker may not move up more than one level .

The space your Worker moves into must be unoccupied (not containing a Worker or Dome).

Build a block ( ) or dome ( ) on an unoccupied space neighboring the moved Worker.

You can build onto a level of any height, but you

Winning the Game

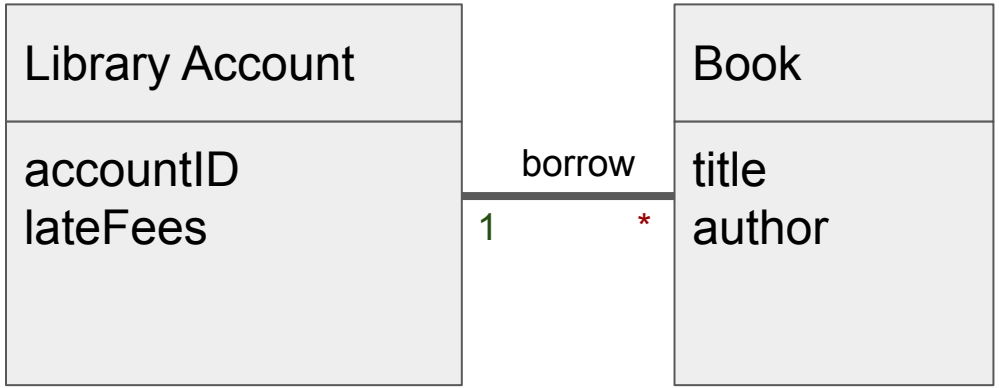
1. If one of your Workers moves up on top of level 3 during your turn, you instantly win!
2. You must always perform a move then build on your turn. If you are unable to, you lose.

Components

Outlook: Low Representational Gap

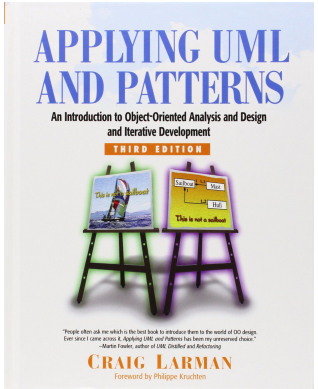
Identified concepts provide inspiration for classes in the implementation

Classes mirroring domain concepts often intuitive to understand
(low representational gap)




```
class Account {
    id: Int;
    lateFees: Int;
    borrowed: List<Book>;
    boolean borrow(Book) { … }
    void save();
}
class Book { … }
```

System Sequence Diagram



Chapter 10

Understanding system behavior

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**

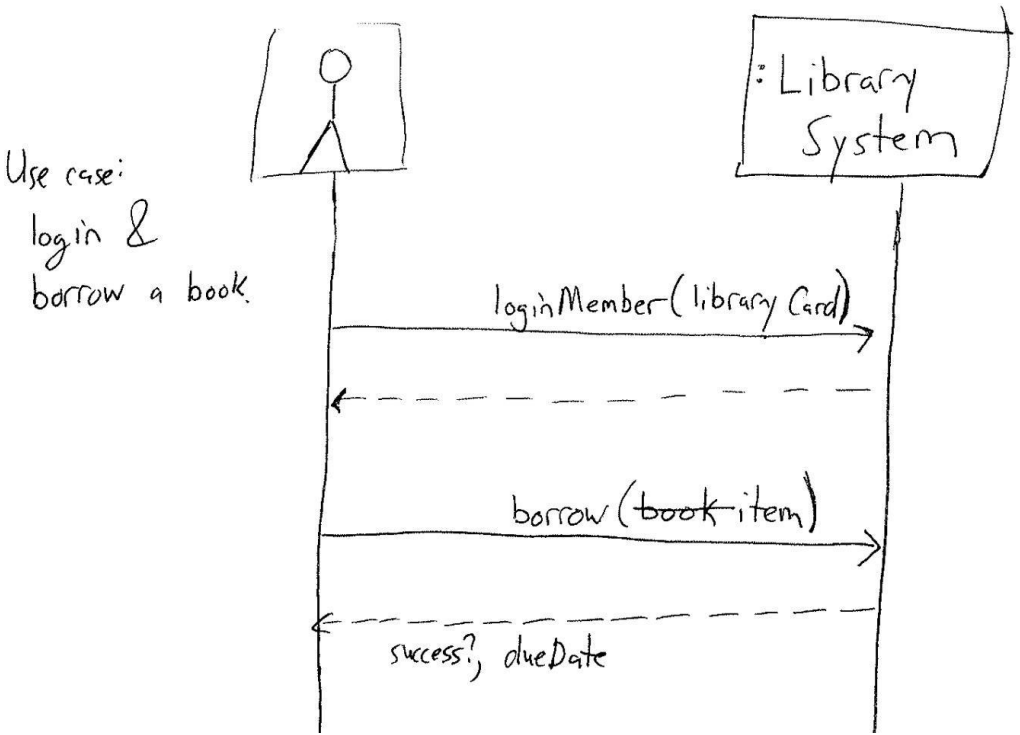
- System-level components only: e.g., A user and the overall system

One example for the library system

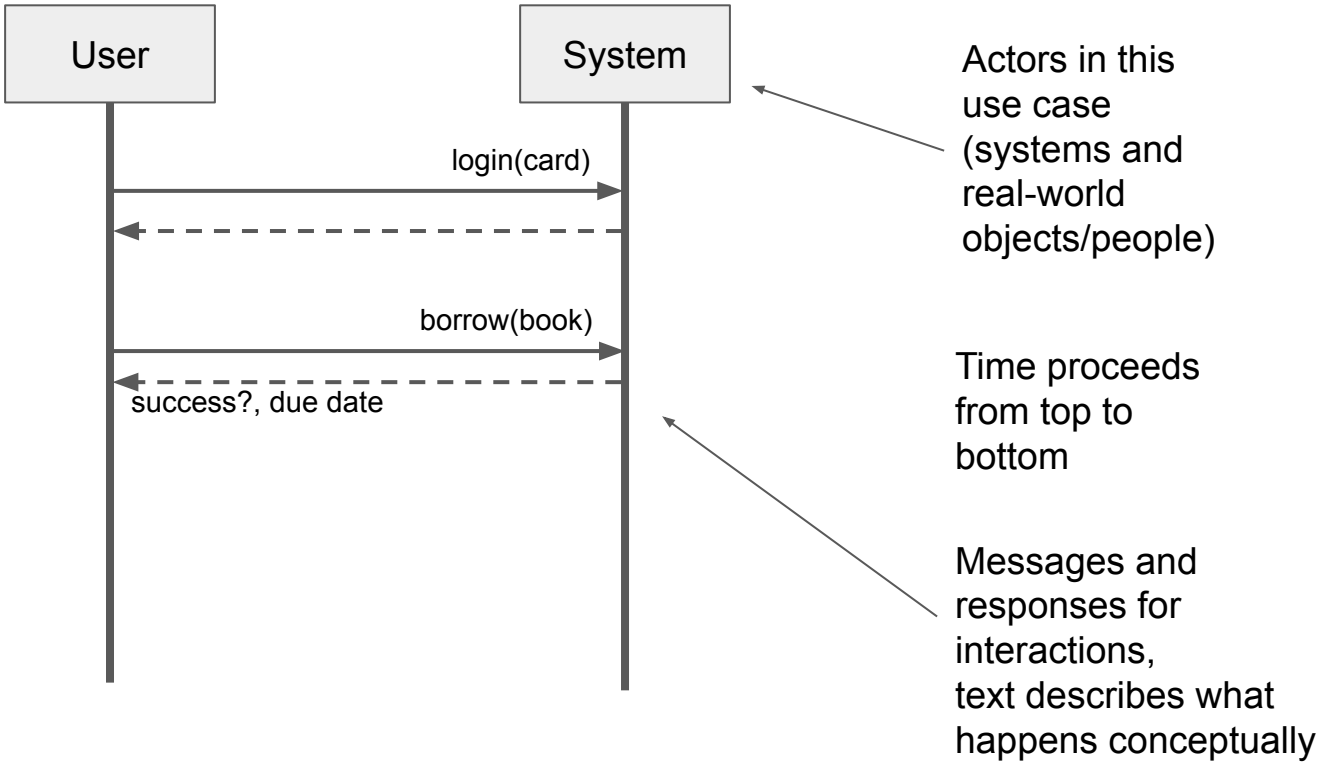
Use case scenario: A library member should be able to use her library card to log in at a library system kiosk and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its rental period to the current day, and record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

One example 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.

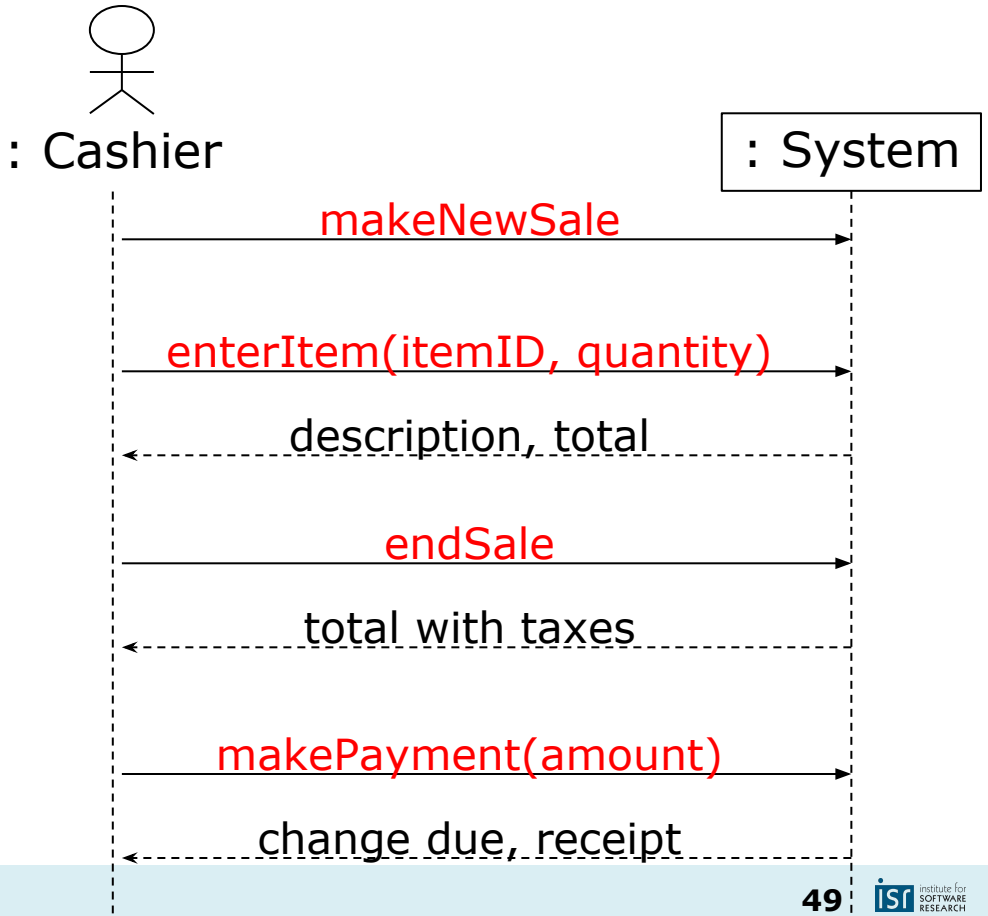


UML Sequence Diagram Notation

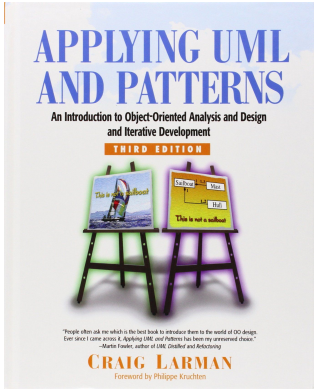


Outlook: System Sequence Diagrams to Tests

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



Behavioral Contracts

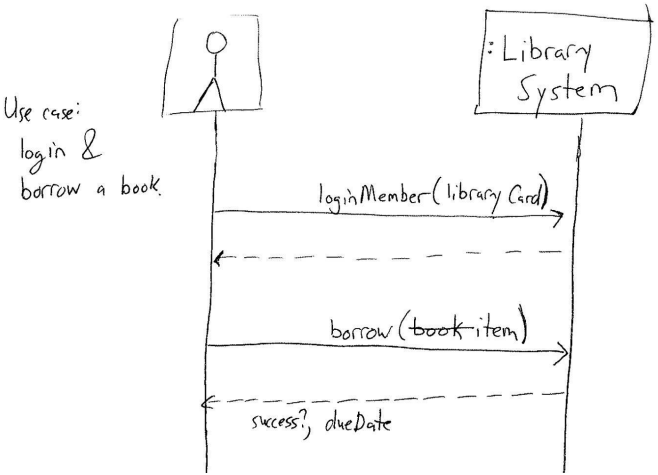


Chapter 11

Formalize system at boundary

A *system behavioral contract* describes the pre-conditions and post-conditions for some operation identified in the system sequence diagrams

- System-level textual specifications, like software specifications



System behavioral contract example

Operation: borrow(item)

Pre-conditions: Library member has already logged in to the system.

Item is not currently borrowed by another member.

Post-conditions: Logged-in member's account records the newly-borrowed item, or the member is warned she has an outstanding late fee.

The newly-borrowed item contains a future due date, computed as the item's rental period plus the current date.

Distinguishing domain vs. implementation concepts

Distinguishing domain vs. implementation concepts

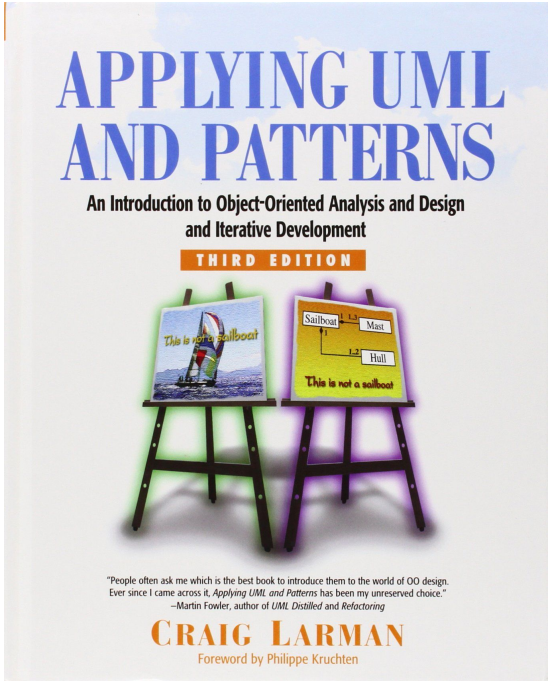
- Domain-level concepts:
  - Almost anything with a real-world analogue
- Implementation-level concepts:
  - Implementation-like method names
  - Programming types
  - Visibility modifiers
  - Helper methods or classes
  - Artifacts of design patterns

Recommended Reading: Applying UML and Patterns

Detailed coverage of modeling steps

Explains UML notation

Many examples



Chapter 9

Summary: Understanding the problem domain

Know your tools to build domain-level representations

- Domain models
- System sequence diagrams
- System behavioral contracts

Be fast and (sometimes) loose

- Elide obvious(?) details
- Iterate, iterate, iterate, …

Get feedback from domain experts

- Use only domain-level concepts

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