Principles of Software Construction:
Objects, Design, and Concurrency

Part 2: Designing (Sub-)Systems

Object-oriented analysis: Modeling a problem domain

Charlie Garrod **Bogdan Vasilescu**

School of Computer Science
institute for SOFTWARE RESEARCH

Intro to Java

Git, CI

UML

Static Analysis

Performance

GUIs

GUIs

More Git

Design

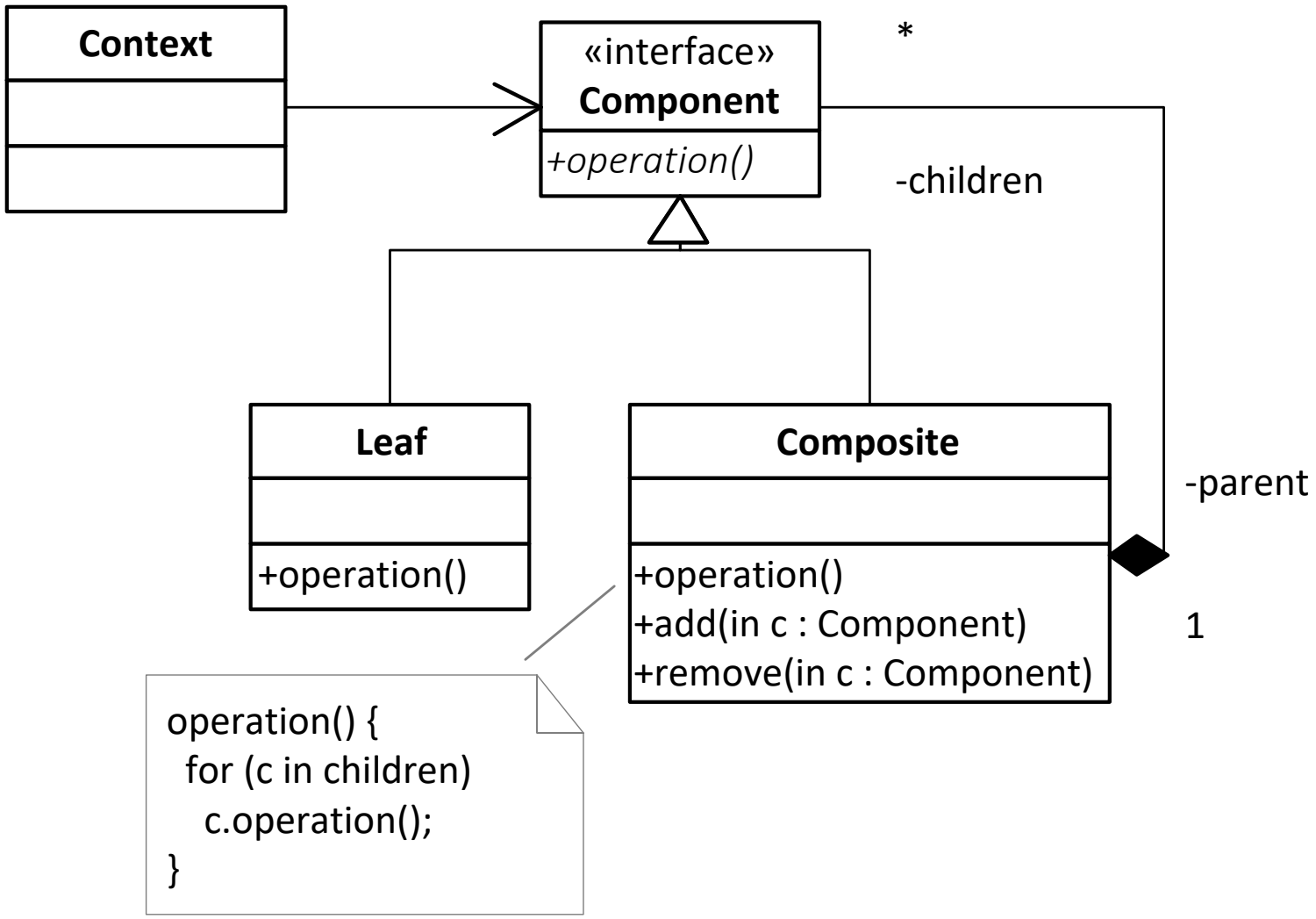
Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	
	Design for Reuse at Scale: Frameworks and APIs	

Administrivia

- Homework 3 due tonight 11:59 p.m.
  - Homework 4 out soon
- (Optional) reading for today:
  - UML & Patterns Ch 17: Use case realizations, interaction diagrams (POS example)
  - EJ 49, 54, 69: Check parameters for validity, return empty not null, use exceptions for exceptional conditions
- Required reading for next Tuesday:
  - UML & Patterns Ch 14—16: More interaction diagrams, responsibility assignment
- **Midterm exam Thursday next week (Feb 15)**
  - **Review session: Wednesday Feb 14 5-7pm Margaret Morrison A14**
  - Practice exam coming soon

Key concepts from Tuesday

The Composite Design Pattern

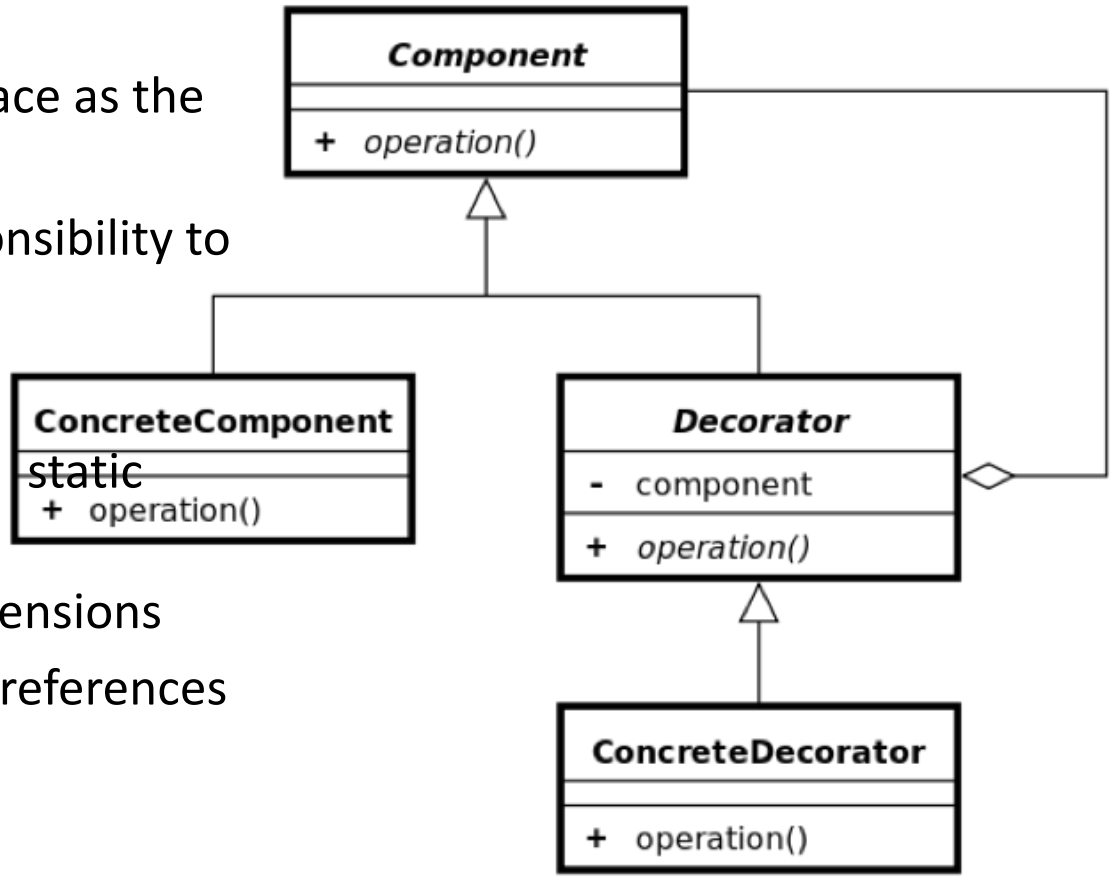


Review

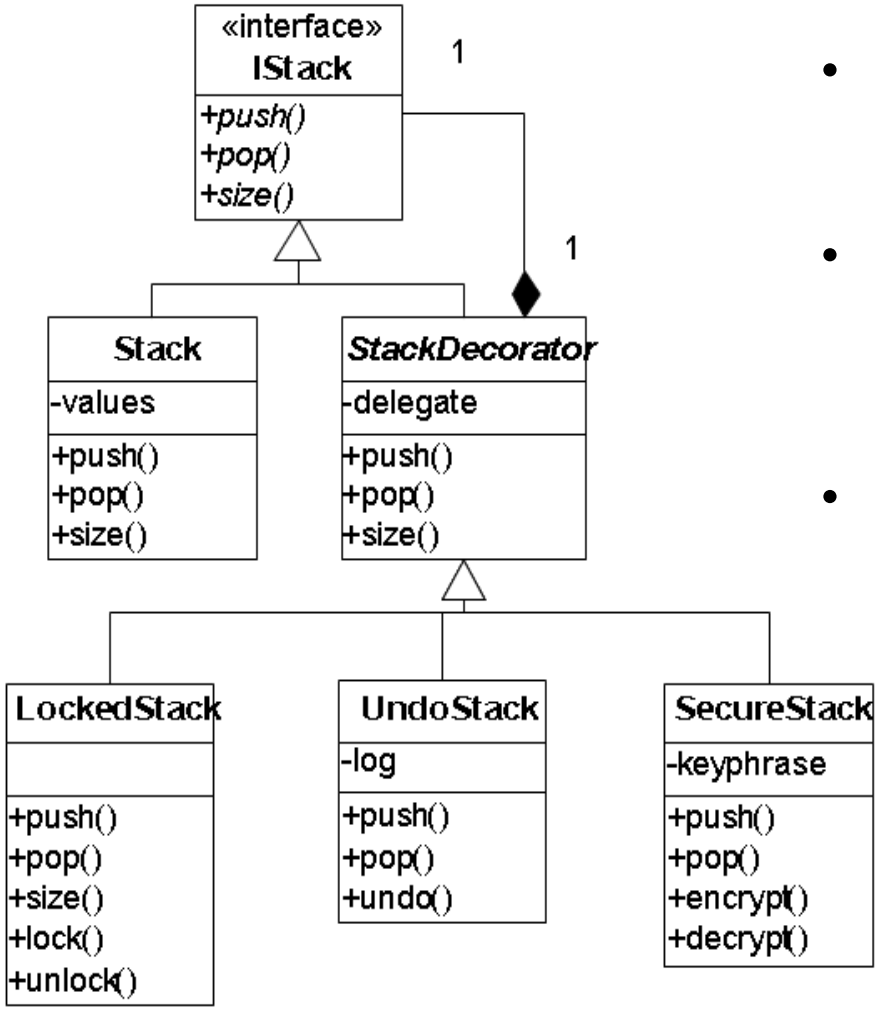
The Decorator Design Pattern

Review

- **Problem**: Need arbitrary / dynamically composable extensions to individual objects.
- **Solution**:
  - Implement common interface as the object you are extending
  - But delegate primary responsibility to an underlying object.
- **Consequences**:
  - More flexible than inheritance
  - Customizable, cohesive extensions
  - Breaks object identity, self-references



Using the Decorator for our Stack example



Using the decorator classes

- To construct a plain stack:

```
Stack s = new Stack();
```

- To construct an plain undo stack:

```
UndoStack s = new
    UndoStack(new Stack());
```

- To construct a secure synchronized undo stack:

```
SecureStack s = new
    SecureStack(new
    SynchronizedStack(new
    UndoStack(new Stack())));
```

Review

Today

- Design goals and design principles

Metrics of software quality

Source: Braude, Bernstein, Software Engineering. Wiley 2011

- Sufficiency / functional correctness
  - Fails to implement the specifications … Satisfies all of the specifications
- Robustness
  - Will crash on any anomalous event … Recovers from all anomalous events
- Flexibility
  - Must be replaced entirely if spec changes … Easily adaptable to changes
- Reusability
  - Cannot be used in another application … Usable without modification
- Efficiency
  - Fails to satisfy speed or storage requirement … satisfies requirements
- Scalability
  - Cannot be used as the basis of a larger version … is an outstanding basis…
- Security
  - Security not accounted for at all … No manner of breaching security is known

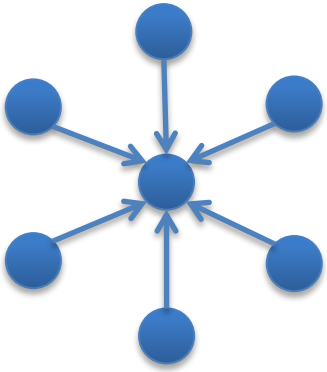
Design challenges/goals

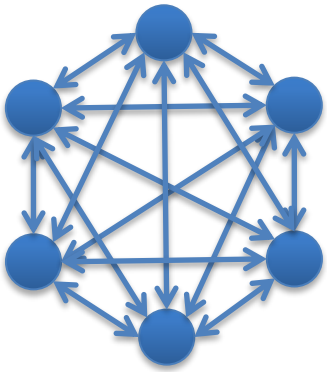
Design principles

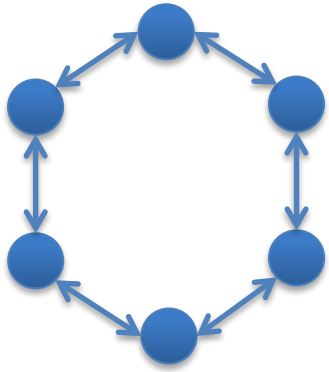
- Low coupling
- Low representational gap
- High cohesion

A design principle for reuse: *low coupling*

- Each component should depend on as few other components as possible







- Benefits of low coupling:
  - Enhances understandability
  - Reduces cost of change
  - Eases reuse

High Coupling is undesirable

- Element with low coupling depends on only few other elements (classes, subsystems, …)
  - “few" is context-dependent
- A class with high coupling relies on many other classes
  - Changes in related classes force local changes; changes in local class forces changes in related classes (brittle, rippling effects)
  - Harder to understand in isolation.
  - Harder to reuse because requires additional presence of other dependent classes
  - Difficult to extend – changes in many places

Which classes are coupled?
How can coupling be improved?

```
class Shipment {
        private List<Box> boxes;
        int getWeight() {
                int w=0;
                for (Box box: boxes)
                        for (Item item: box.getItems())
                                w += item.weight;
                return w;
}
class Box {
        private List<Item> items;
        Iterable<Item> getItems() { return items;}
}
class Item {
        Box containedIn;
        int weight;
}
```

A different design.
How can coupling be improved?

```
class Box {
        private List<Item> items;
        private Map<Item,Integer> weights;
        Iterable<Item> getItems() { return items;}
        int getWeight(Item item) { return weights.get(item);}
}
class Item {
        private Box containedIn;
        int getWeight() { return containedIn.getWeight(this);}
}
```

Law of Demeter

- *Each module should have only limited knowledge about other units: only units "closely" related to the current unit*
- In particular: Don’t talk to strangers!
- For instance, no a.getB().getC().foo()

```
for (Item i: shipment.getBox().getItems())
        i.getWeight() …
```

Coupling: Discussion

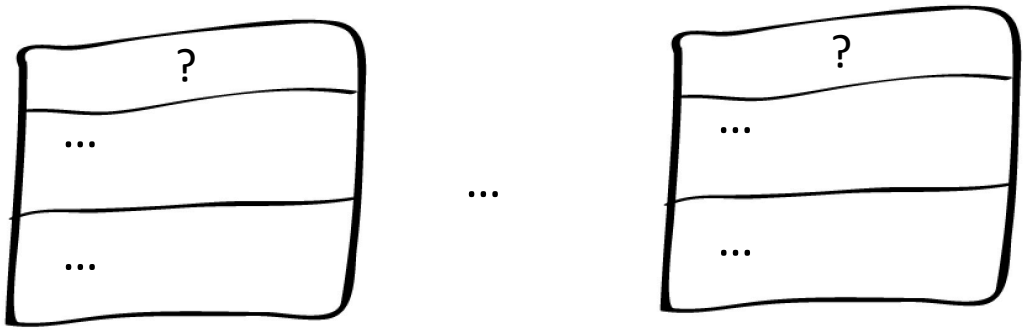
- Subclass/superclass coupling is particularly strong
  - protected fields and methods are visible
  - subclass is fragile to many superclass changes, e.g. change in method signatures, added abstract methods
  - *Guideline: prefer composition to inheritance, to reduce coupling*
- High coupling to very stable elements is usually not problematic
  - A stable interface is unlikely to change, and likely well-understood
  - *Prefer coupling to interfaces over coupling to implementations*
- Coupling is one principle among many
  - Consider cohesion, low repr. gap, and other principles

DESIGN PRINCIPLE: LOW REPRESENTATIONAL GAP

Representational gap

- Real-world concepts:

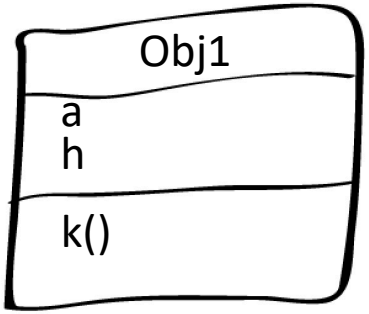
- Software concepts:

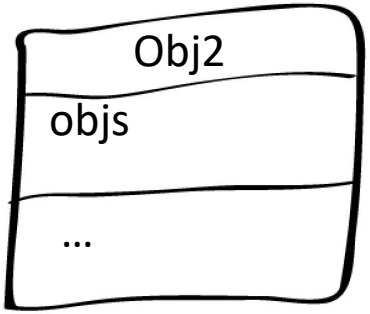


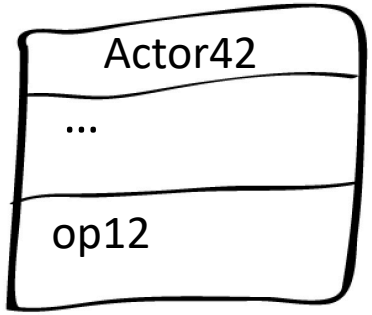
Representational gap

- Real-world concepts:

- Software concepts:



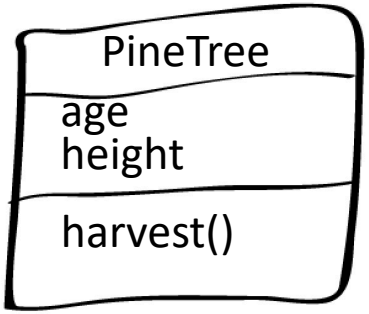


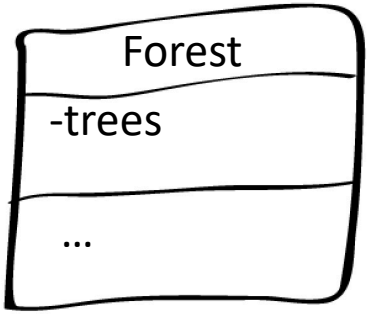


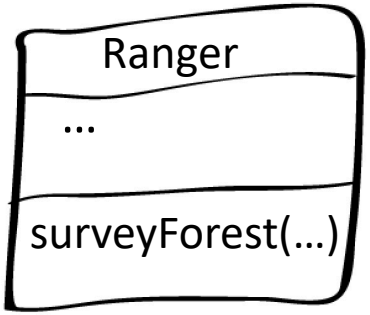
Representational gap

- Real-world concepts:

- Software concepts:





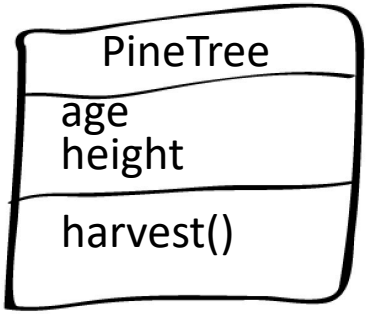


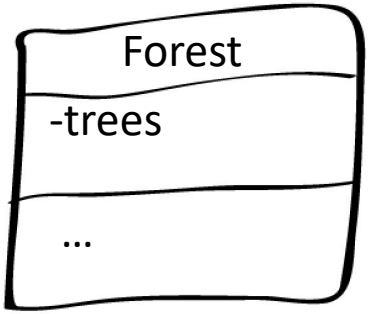
Benefits of low representational gap

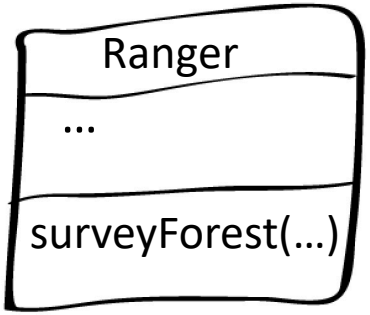
- Facilitates understanding of design and implementation
- Facilitates traceability from problem to solution
- Facilitates evolution

A related design principle: high cohesion

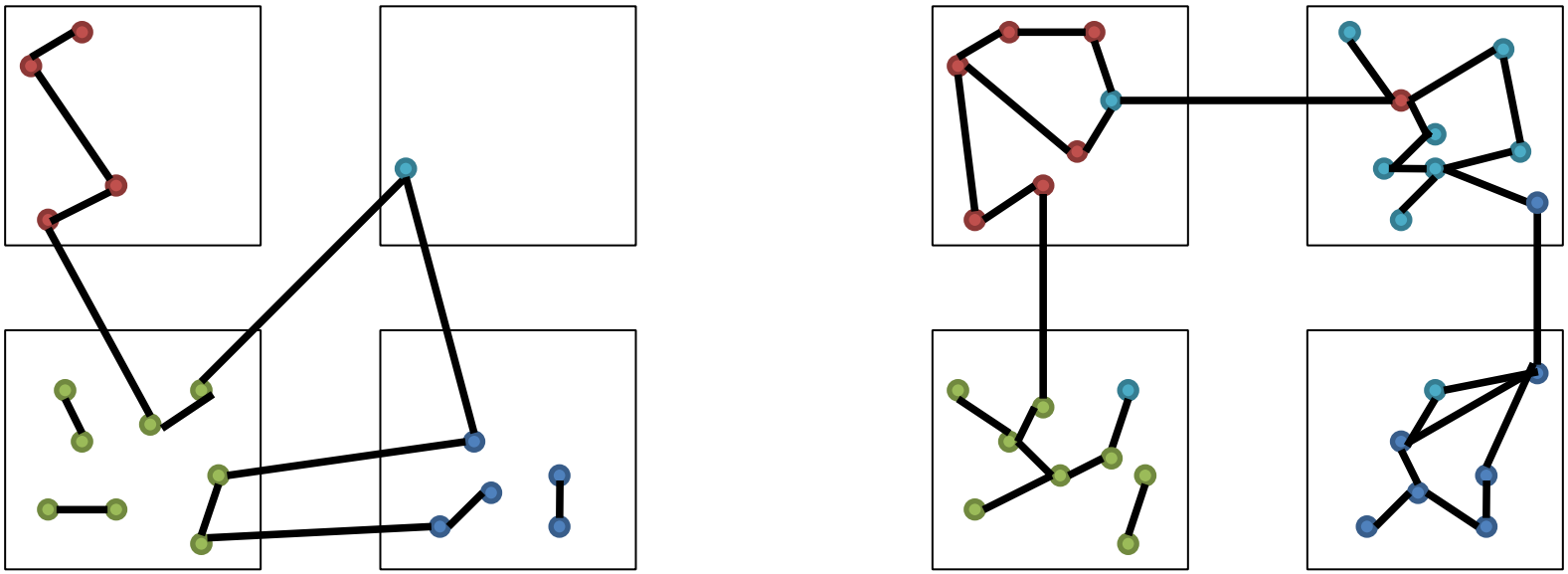
- Each component should have a small set of closely-related responsibilities
- Benefits:
  - Facilitates understandability
  - Facilitates reuse
  - Eases maintenance







High (left) vs low (right) cohesion



```
class DatabaseApplication
        public void authorizeOrder(Data data, User currentUser, ...){
                // check authorization
                // lock objects for synchronization
                // validate buffer
                // log start of operation
                // perform operation
                // log end of operation
                // release lock on objects
        }
        public void startShipping(OtherData data, User currentUser, ...){
                // check authorization
                // lock objects for synchronization
                // validate buffer
                // log start of operation
                // perform operation
                // log end of operation
                // release lock on objects
        }
}
```

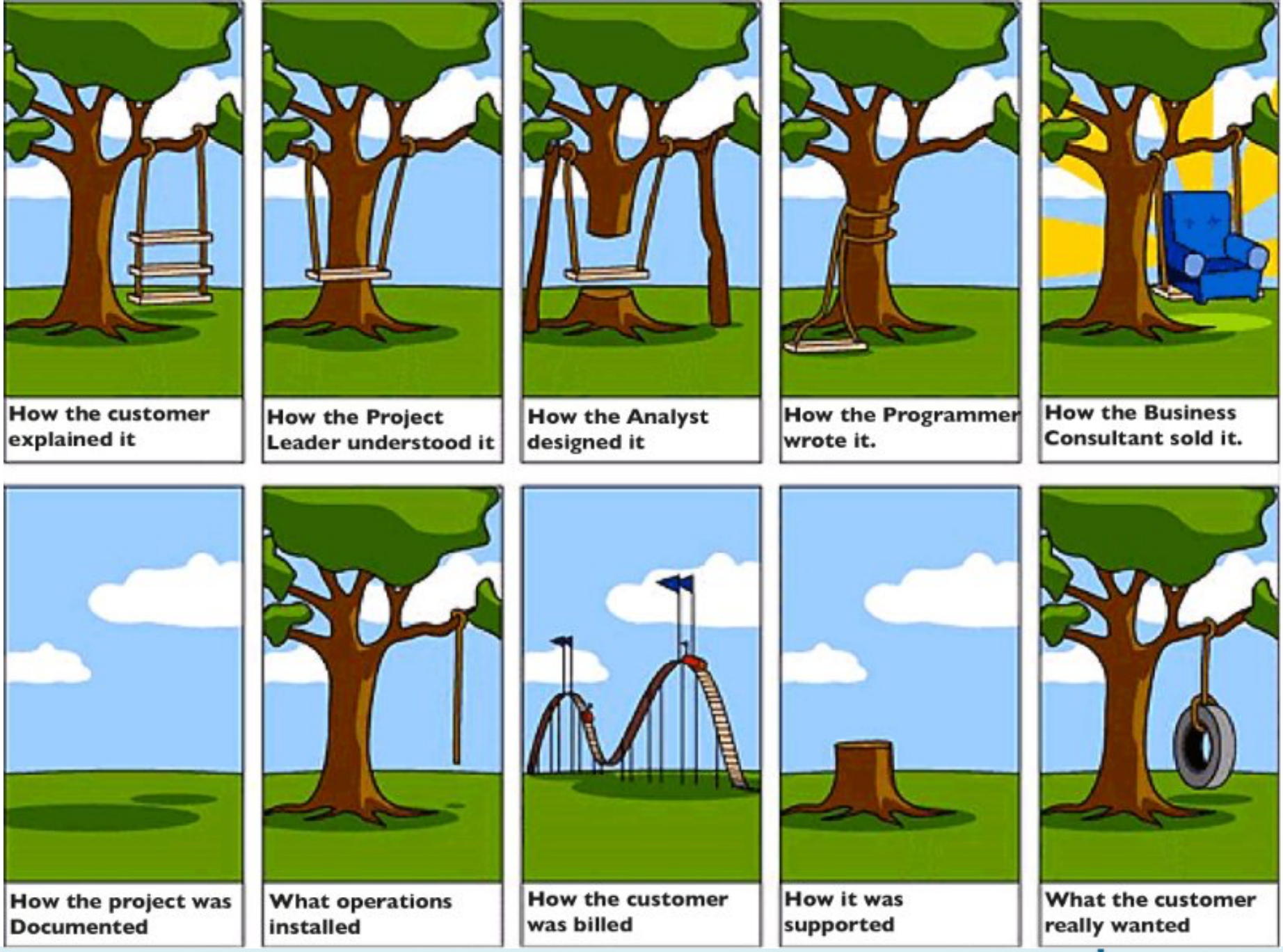
Coupling vs. cohesion

- All code in one component?
  - Low cohesion, low coupling
- Every statement / method in a separate component?
  - High cohesion, high coupling

Summary

- Design principles are useful heuristics
  - Reduce coupling to increase understandability, reuse
  - Lower representational gap to increase understandability, maintainability
  - Increase cohesion to increase understandability

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 sold it.
How the project was Documented
What operations installed
How the customer was billed
How it was supported
What the customer really wanted

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

- What does the customer want?
- What
  constr
- Custo
  biased
- Difficult to define requirements precisely
- (Are we building the right thing? Not: Are we building the thing
  righ

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

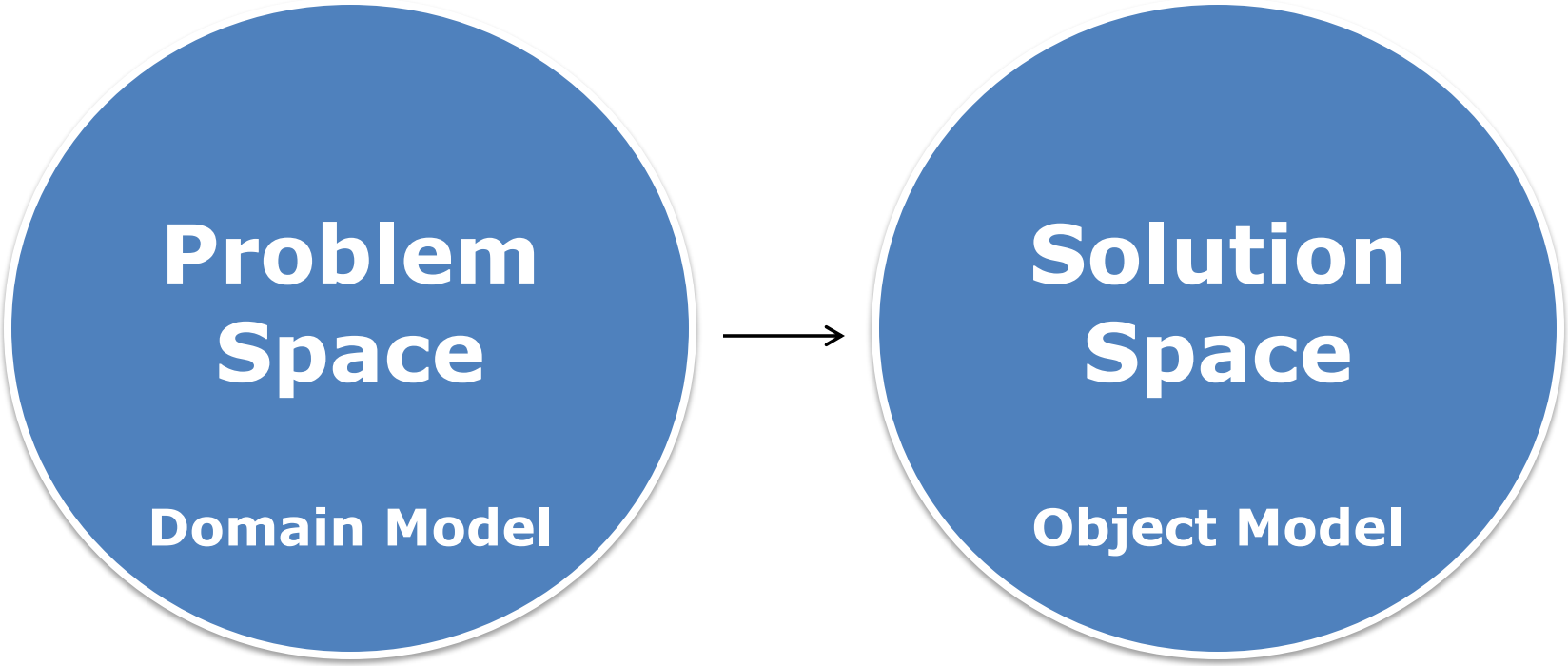
Challenges:
How do we start implementing them?
How do we cope with changes?

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)

Our path toward a more formal design process



- 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

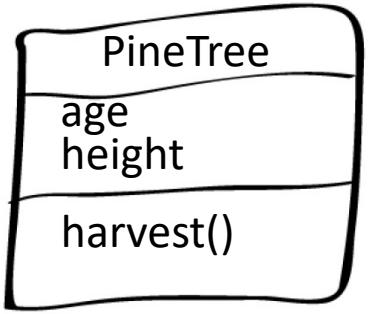
Representational gap

- Real-world concepts:

- Software concepts:

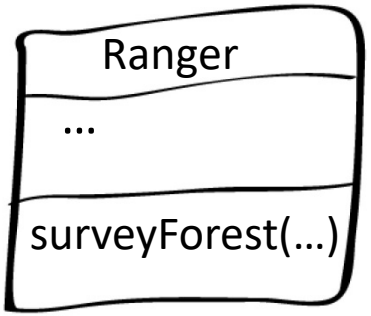
Problem Space

Domain Model



Solution Space

Object Model



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

…

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

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

Understanding the problem

Defining a solution

Input to the design process: Requirements and use cases

- Typically prose:

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 re
is borrowed
member mu
member's li

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

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

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

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.

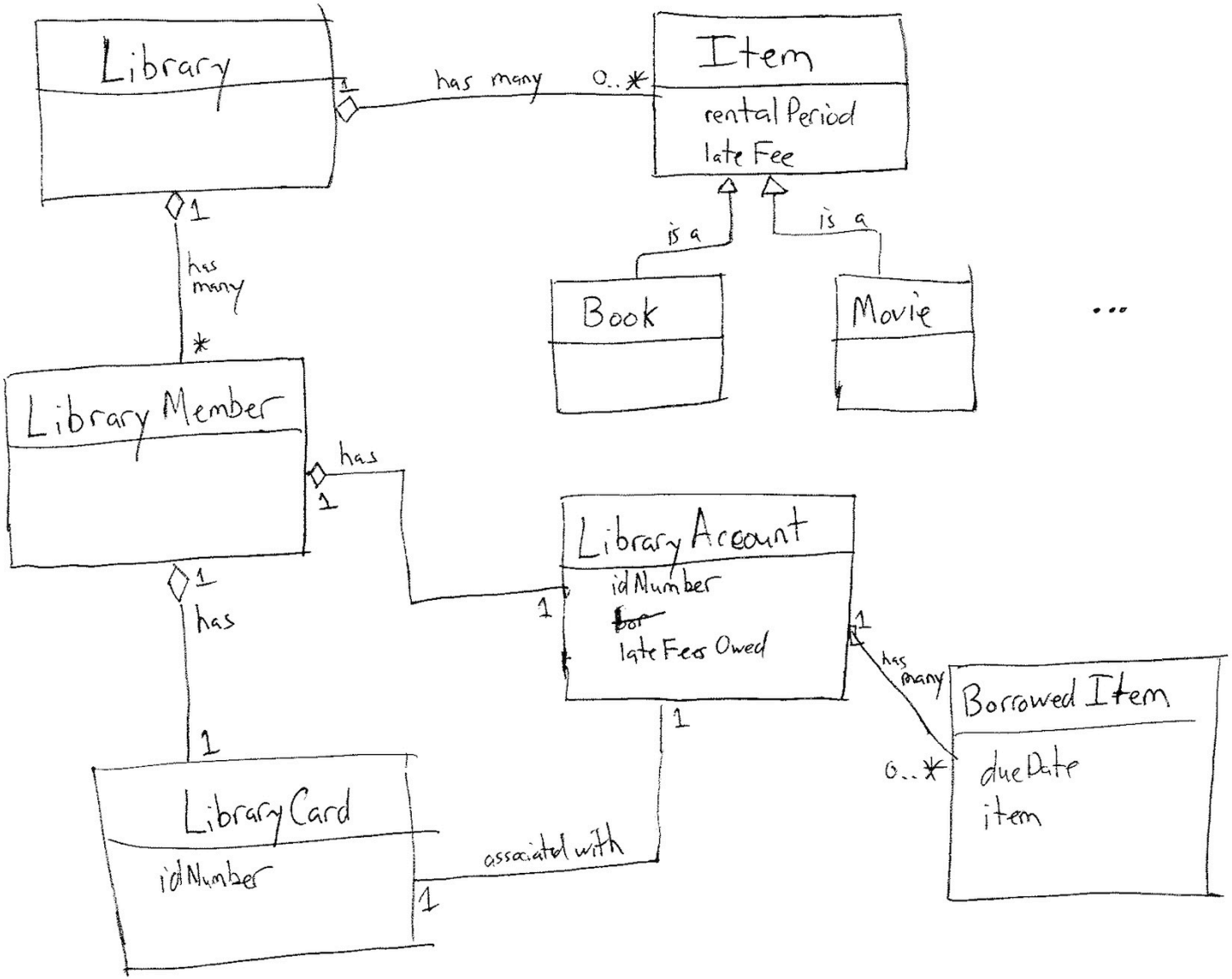
A member's library account records which items the member has borrowed and the due date for each borrowed item. Each type of item has a default rental period, which determines the item's due date when the item is borrowed. If a member returns an item after the item's due date, the member owes a late fee specific for that item, an amount of money recorded in the member's library account.

Read description carefully, look for nouns and verbs

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



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

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

Build a domain model for Monopoly

Build a domain model for Monopoly

Monopoly is a game in which each player has a piece that moves around a game board, with the piece's change in location determined by rolling a pair of dice. The game board consists of a set of properties (initially owned by a bank) that may be purchased by the players.

When a piece lands on a property that is not owned, the player may use money to buy the property from the bank for that property's price. If a player lands on a property she already owns, she may build houses and hotels on the property; each house and hotel costs some price specific for the property. When a player's piece lands on a property owned by another player, the owner collects money (rent) from the player whose piece landed on the property; the rent depends on the number of houses and hotels built on the property.

The game is played until only one remaining player has money and property, with all the other players being bankrupt.

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"

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