Principles of Software Construction: Objects, Design, and Concurrency

Responsibility Assignment

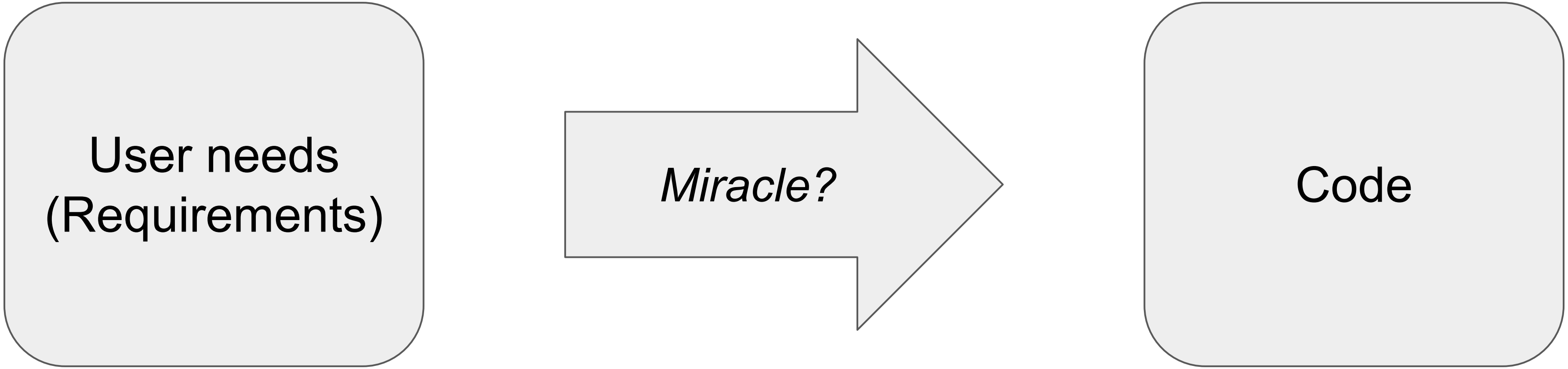
Claire Le Goues **Vincent Hellendoorn**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrative notes

- Midterm 1 next week Thursday
  - Sample questions out shortly
  - Come to OH for help
- Homework 3 (Santorini game engine) out
  - *Must* start early
- Recitation this week: UML design diagrams
- Github actions + Java 18 – not a thing, apparently!


User needs
(Requirements)
Miracle?
Code

Problem Space

(Domain Model)

→

Solution Space

(Object Model)

- 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**

Last Class

OO Analysis: Understanding the problem

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

Today

OO Design: Defining a solution

Learning Goals

- UML in the Solution Space
  - Object diagrams: from concepts to classes
  - Interaction diagrams: interactions beyond the system boundary
- Making Design Decisions
  - Expand our vocabulary of **principles**, **patterns** and **heuristics**
  - Apply GRASP patterns to assign responsibilities in designs
  - Reason about tradeoffs among designs
  - Discuss tradeoffs in terms of coupling and cohesion

Testing Retrospectives

- How did it go?

- A few reflections:
  - Ambiguity in specifications
  - Keeping tests simple when using other classes
  - Branch coverage with no asserts??

Convenient Tangent: Integration Testing

```
@Test
public void testCueCards() {
    CardDeck deck = new CardDeck(
        Set.of(new FlashCard("a", "b"), new FlashCard("c", "d")),
        new NonRepeatingOrganizer());
    System.setIn(new Byte…Stream("b\nwrong".getBytes());
    new UI().studyCards(deck);
}
```

Amazing coverage with one test??

Element	Missed Instructions	Cov.	Missed Branches	Cov.
studyCards(CardDeck)		100%		100%
cueCard(FlashCard, Scanner)		100%		100%
cueAllCards(CardDeck, Scanner)		100%		100%
UI()		100%		n/a
Total	0 of 99	100%	0 of 6	100%

Programming a Core Without a GUI/CLI

Is hard! You want to run the code you write.

What *can* you do?

```
function newGame() {
   return newBoard(new Array(9).fill(-1,0,9),0)
}

function newBoard(state, nextPlayer) {
   return {
       play: function(x, y) {
           state[y*3+x] = nextPlayer
           nextPlayer = 1 - nextPlayer
       },
       winner: function() { … }
   }
}
```

Integration Testing

Write tests!

```
g = newGame();
expect(g.winner()).toBe(-1);
g.play(1,1);
expect(g.state[4]).toBe(0);
g.play(...)
...
g.play(...)
expect(g.winner()).toBe(1);
```

```
function newGame() {
   return newBoard(new Array(9).fill(-1,0,9),0)
}

function newBoard(state, nextPlayer) {
   return {
      play: function(x, y) {
         state[y*3+x] = nextPlayer
         nextPlayer = 1 - nextPlayer
      },
      winner: function() { … }
   }
}
```

Integration Testing

- All a GUI does is start the system and send a series of actions
  - You can do that yourself!
- I(&)T tests the collective behavior of a group of units/modules
- Keeping it simple* for now:
  - Design typical gameplay *scenarios*
    - These can correspond directly to your system sequence diagrams!
  - Write tests that exercise each and confirm all expected outcomes →

```
g = newGame();
expect(g.winner()).toBe(-1);
g.play(1,1);
expect(g.state[4]).toBe(0);
g.play(...)
...
g.play(...)
expect(g.winner()).toBe(1);
```

*Technically we should discuss the difference between this and System testing, but we probably won't in this course.

Today

Modeling Implementations with UML

A Word on UML

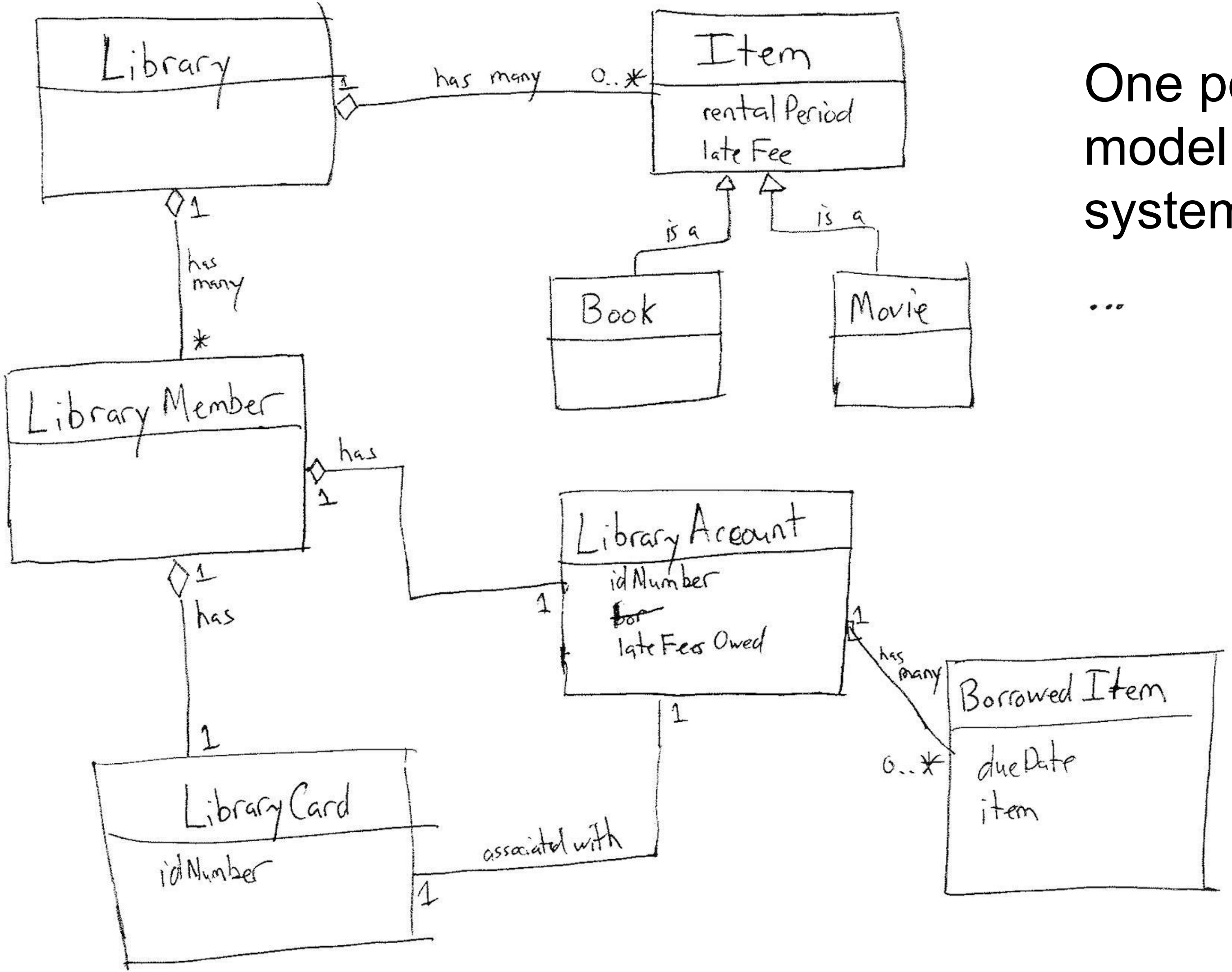
UML is a standard, established notation

Most software engineers can read it, many tools support it

Few practitioners use it rigorously

Commonly used *informally* for sketching, communication, documentation, wall art

In this course: Use UML for communication; follow notation somewhat rigorously, but won't care about all details

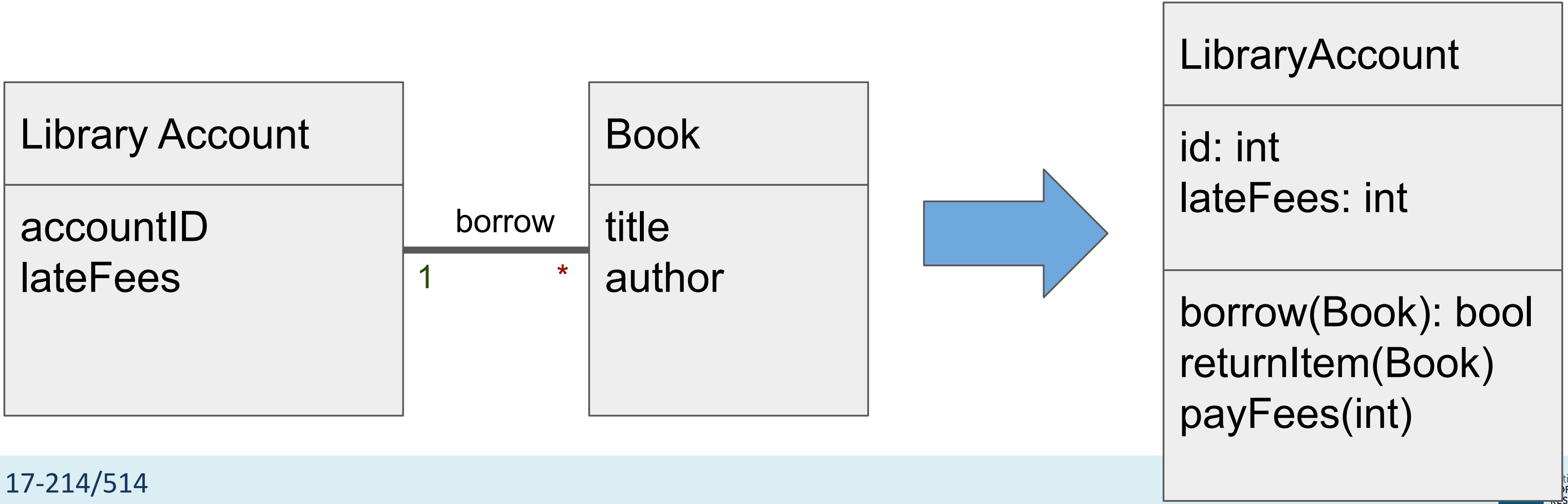


One possible domain model for the library system

...

From concepts to objects

- How are domain concepts different from classes?
  - Should every concept become a class?
  - Does every class need to represent a concept?

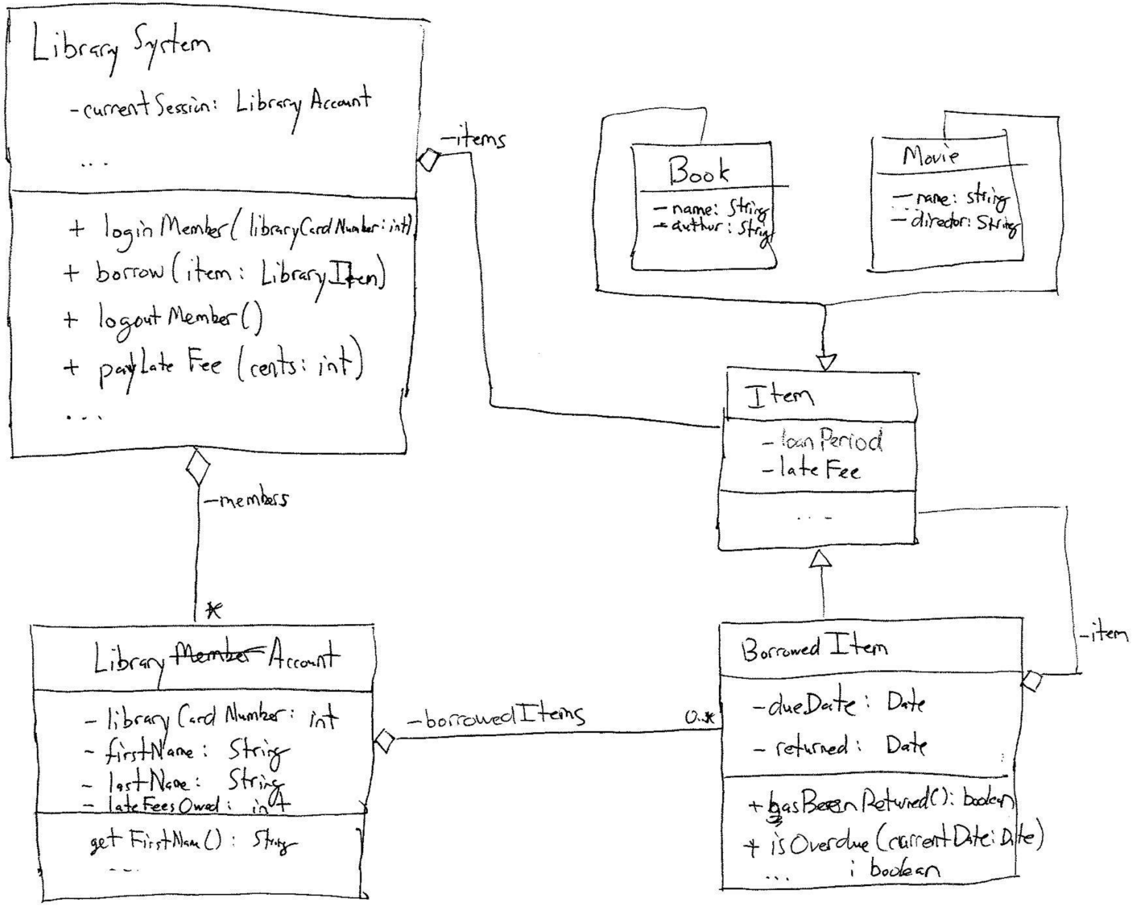


Object Diagrams

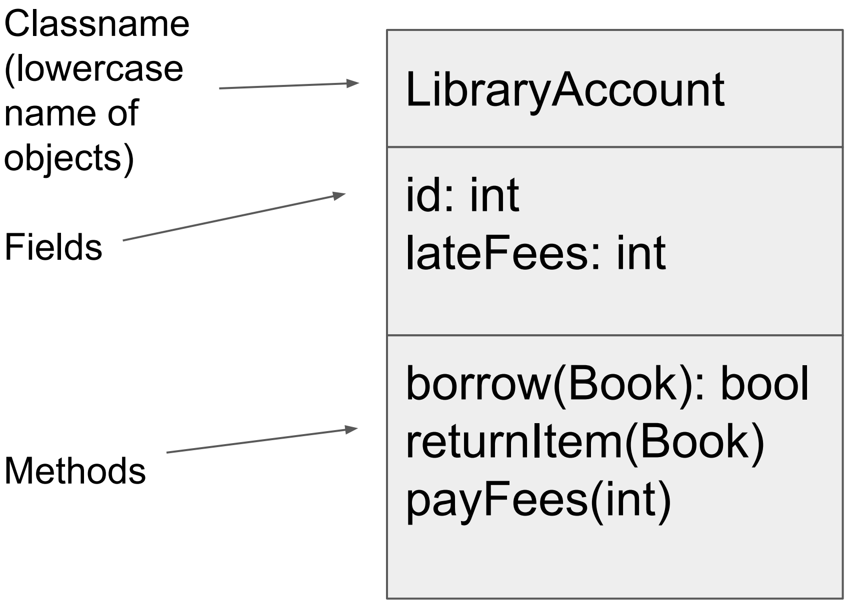
Objects/classes with fields and methods

Interfaces with methods

Associations, visibility, types

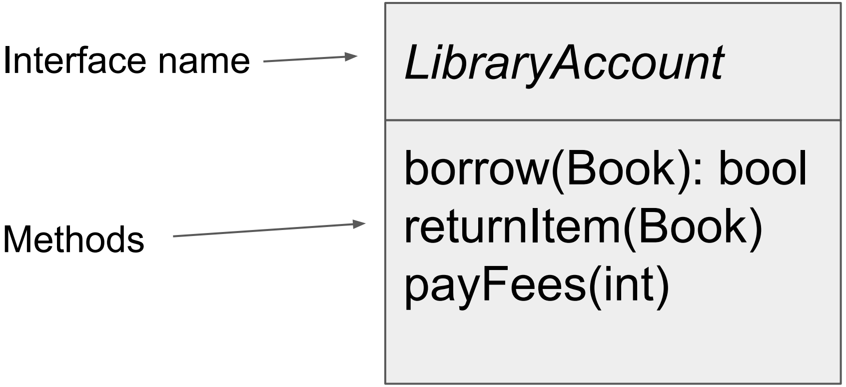


Object Diagram Notation: Classes/Objects



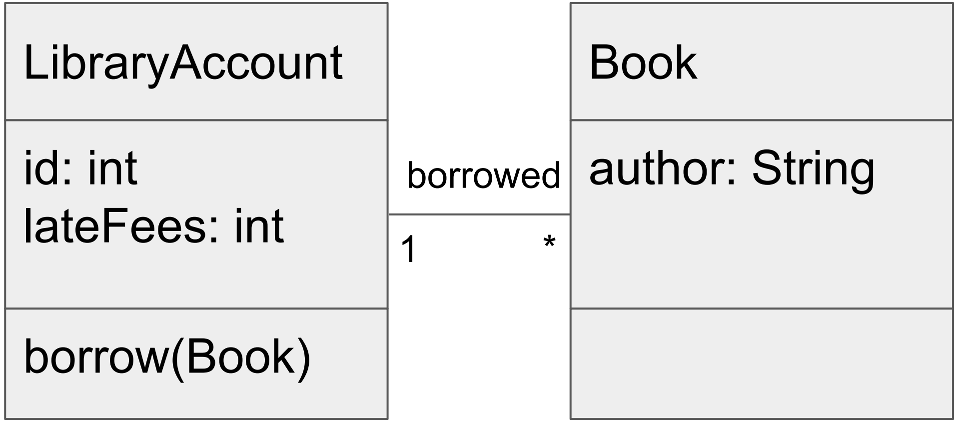

```
class LibraryAccount {
    id: int;
    lateFees: int;
    boolean borrow(Book b) {…}
    void returnItem(Book b) {…}
    void payFees(int payment) {…}
}
```

Object Diagram Notation: Interfaces



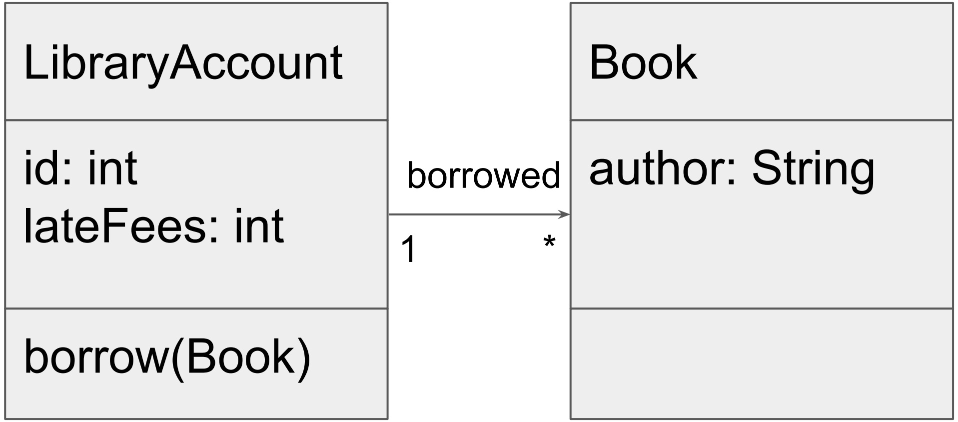

```
interface LibraryAccount {
    boolean borrow(Book b);
    void returnItem(Book b);
    void payFees(int payment);
}
```

Object Diagram Notation: Associations



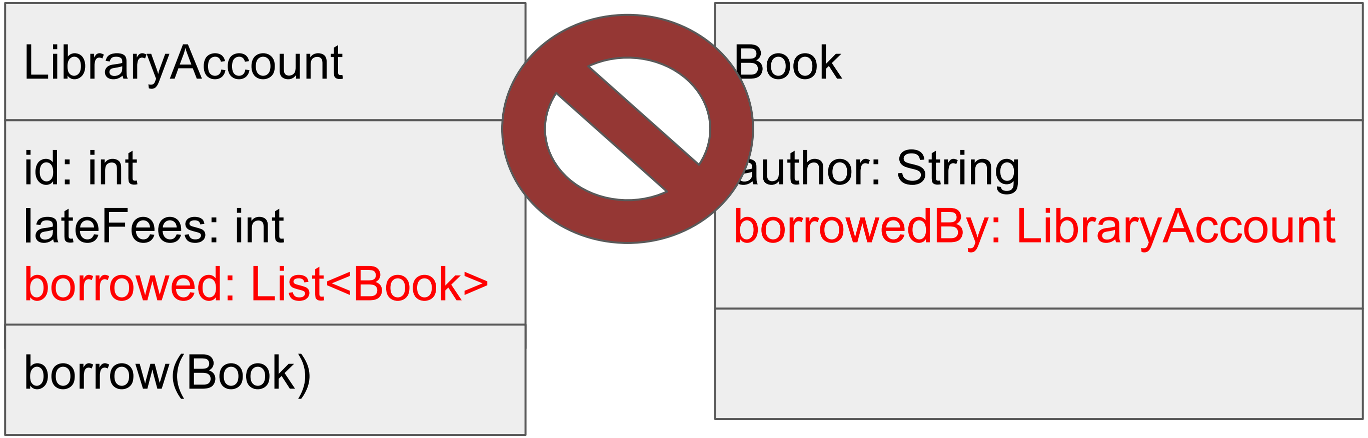

```
class LibraryAccount {
    ...
    List<Book> borrowedBooks;
}
class Book {
    ...
    LibraryAccount borrowedBy;
}
```

Object Diagram Notation: Associations




```
class LibraryAccount {
    ...
    List<Book> borrowedBooks;
}
class Book {
    ...
}
```

Object Diagram Notation: Associations



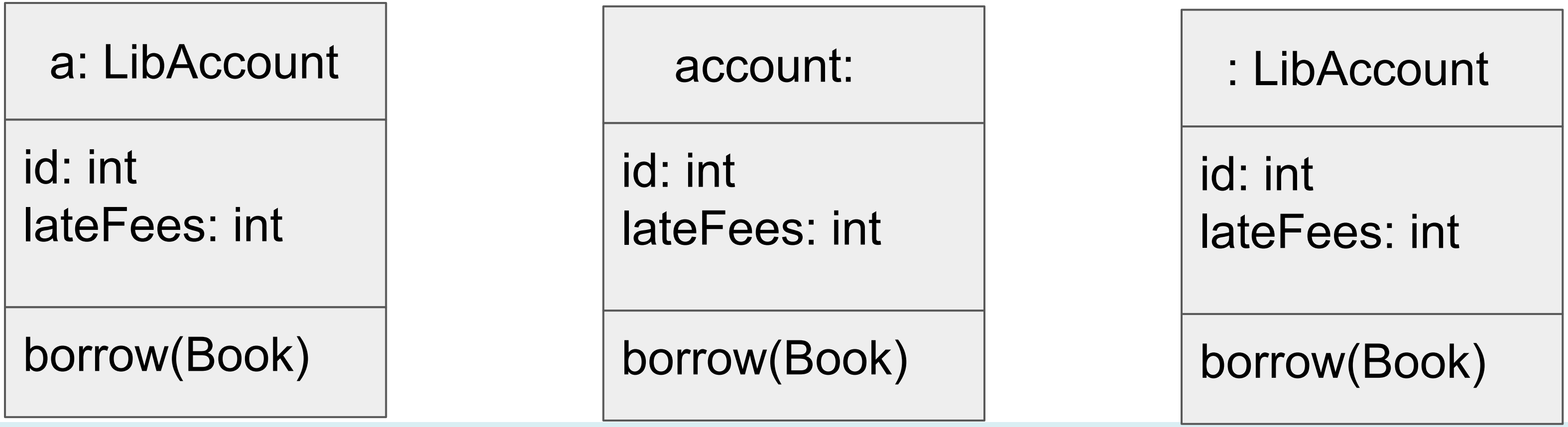
Don’t use fields instead or in addition to associations. Use fields only for basic types

Class Diagram vs Object Diagram

Can model both classes and objects

Terms often used interchangeably

If specific objects should be modeled use “objectId: Class” notation



Class Diagrams and JavaScript/TypeScript

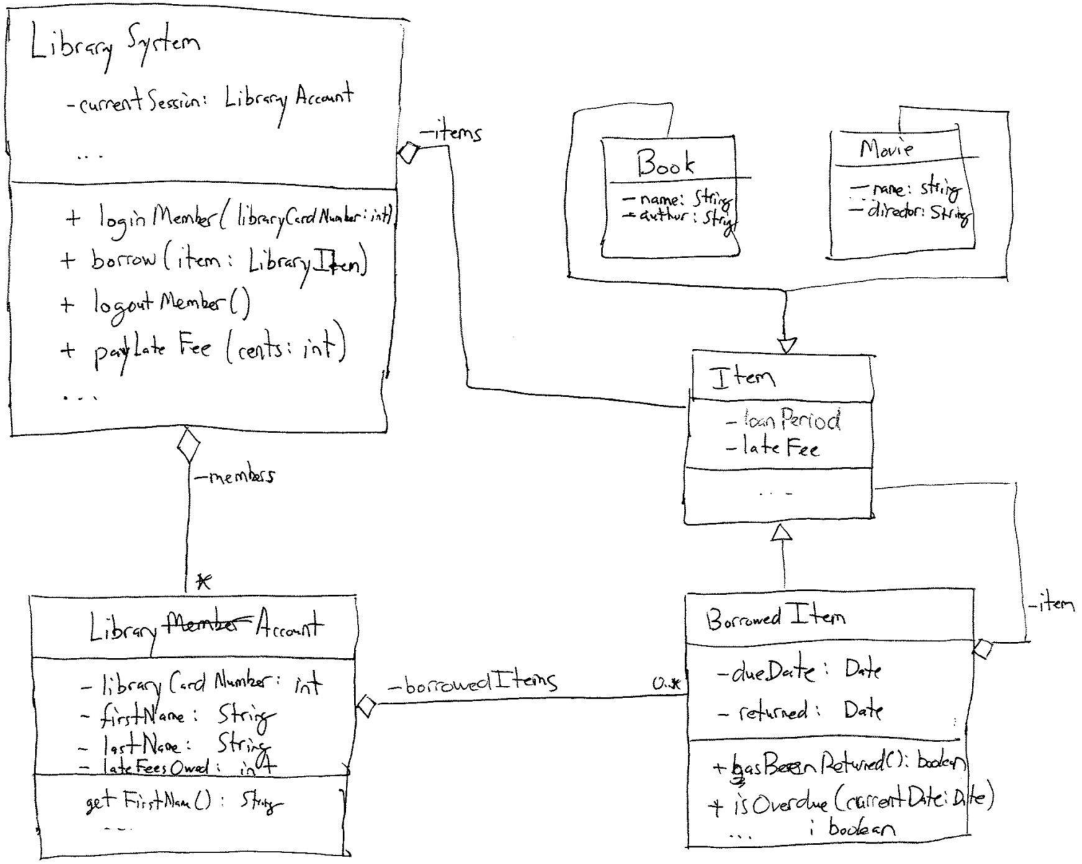
Even when not using classes, use the notation for representing the same idea: many objects sharing a shape

TypeScript interfaces match to class diagram notation

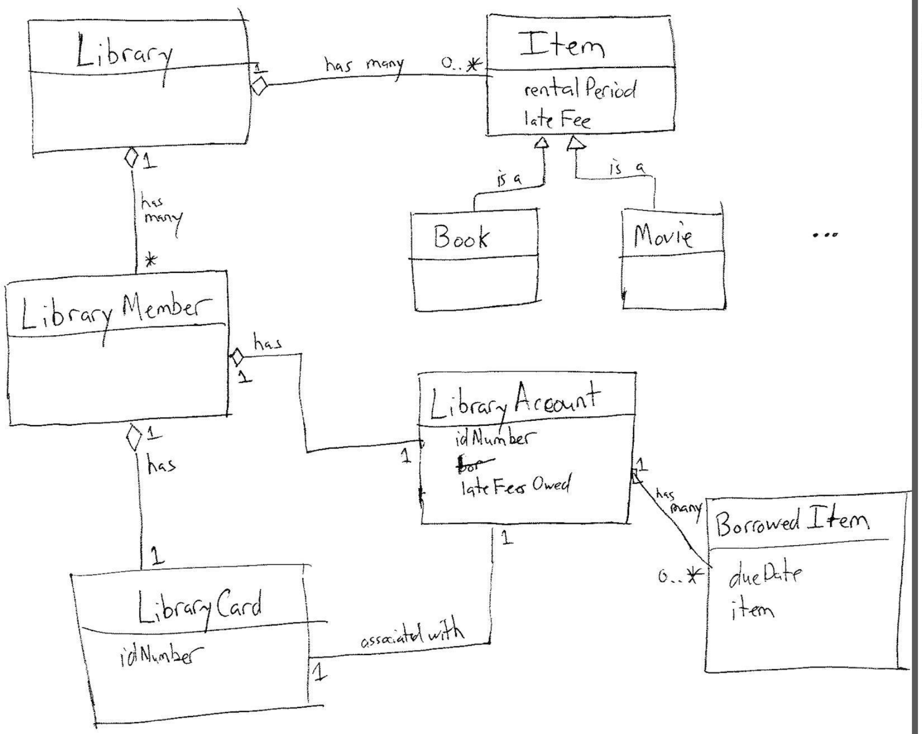
LibraryAccount
id: int lateFees: int
borrow(Book): bool returnItem(Book) payFees(int)

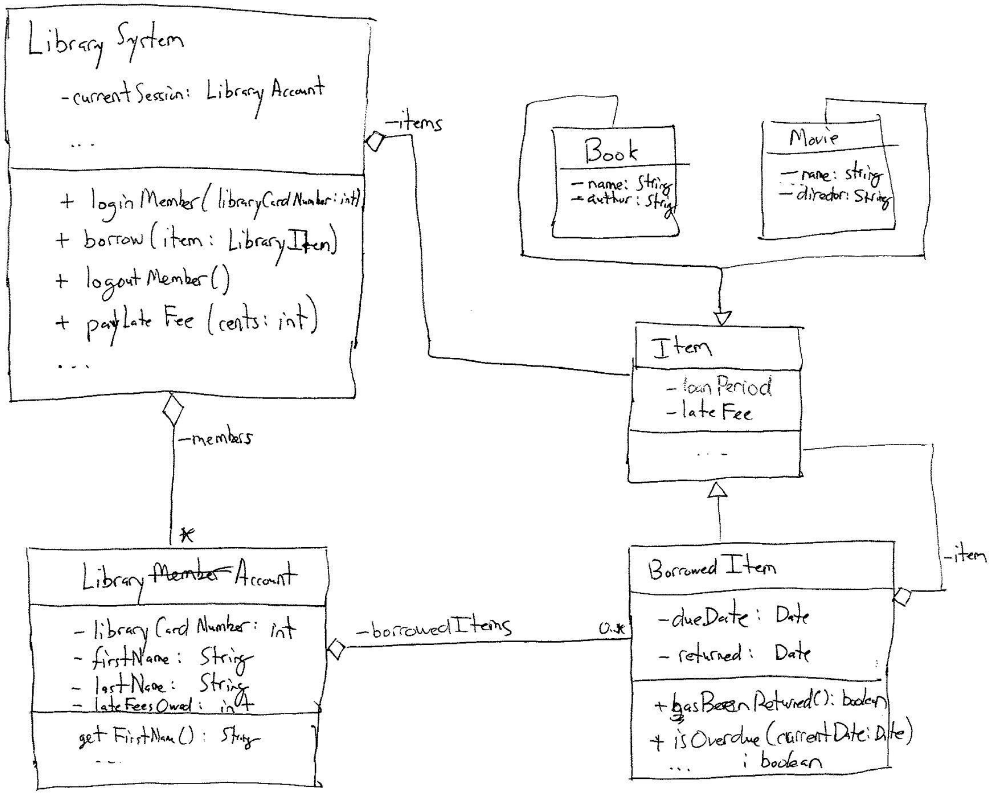
```
function newLibraryAccount(id, lateFees) {
    return {
        borrow: function(book) {…},
        returnItem: function(book) {…},
        payFees: function(payment) {…}
    }
}
```

One object model for the library system



Domain model (left) vs object model (right)





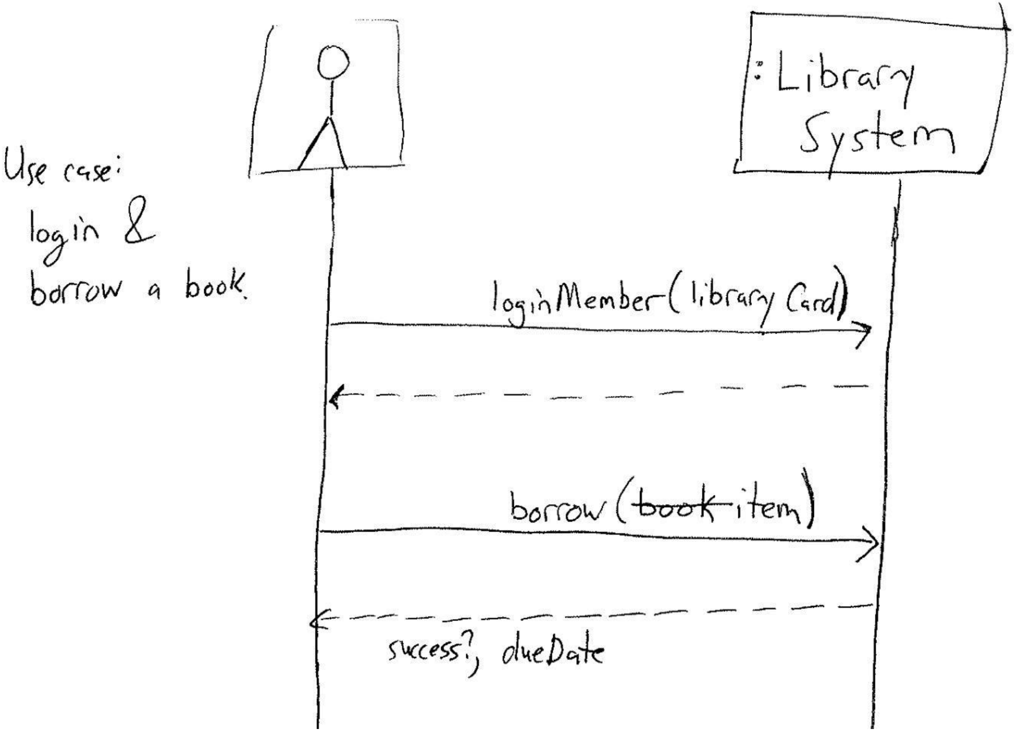
Object diagram notation requirements

We won't be very picky on notation, but:

- Use boxes with 2 or 3 parts for fields, methods as appropriate for classes/objects, interfaces, concepts
- Include types for fields and methods
- Use associations, not fields, where appropriate
- Use association names and cardinalities (we don't care about arrow types, except "is-a")

Modeling interactions past the system boundary

Use case scenario: A library member should be able to <u>use her library card to log in</u> at a library system kiosk and <u>borrow a book</u>. After confirming that the member has no unpaid late fees, the library system should determine the book's <u>due date</u> 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.

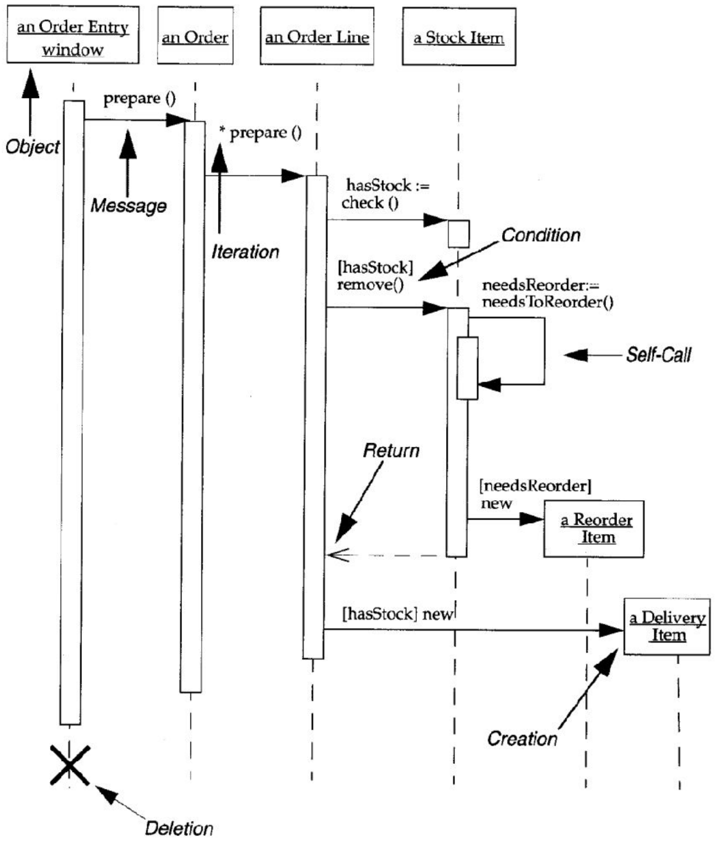


Interaction Diagrams

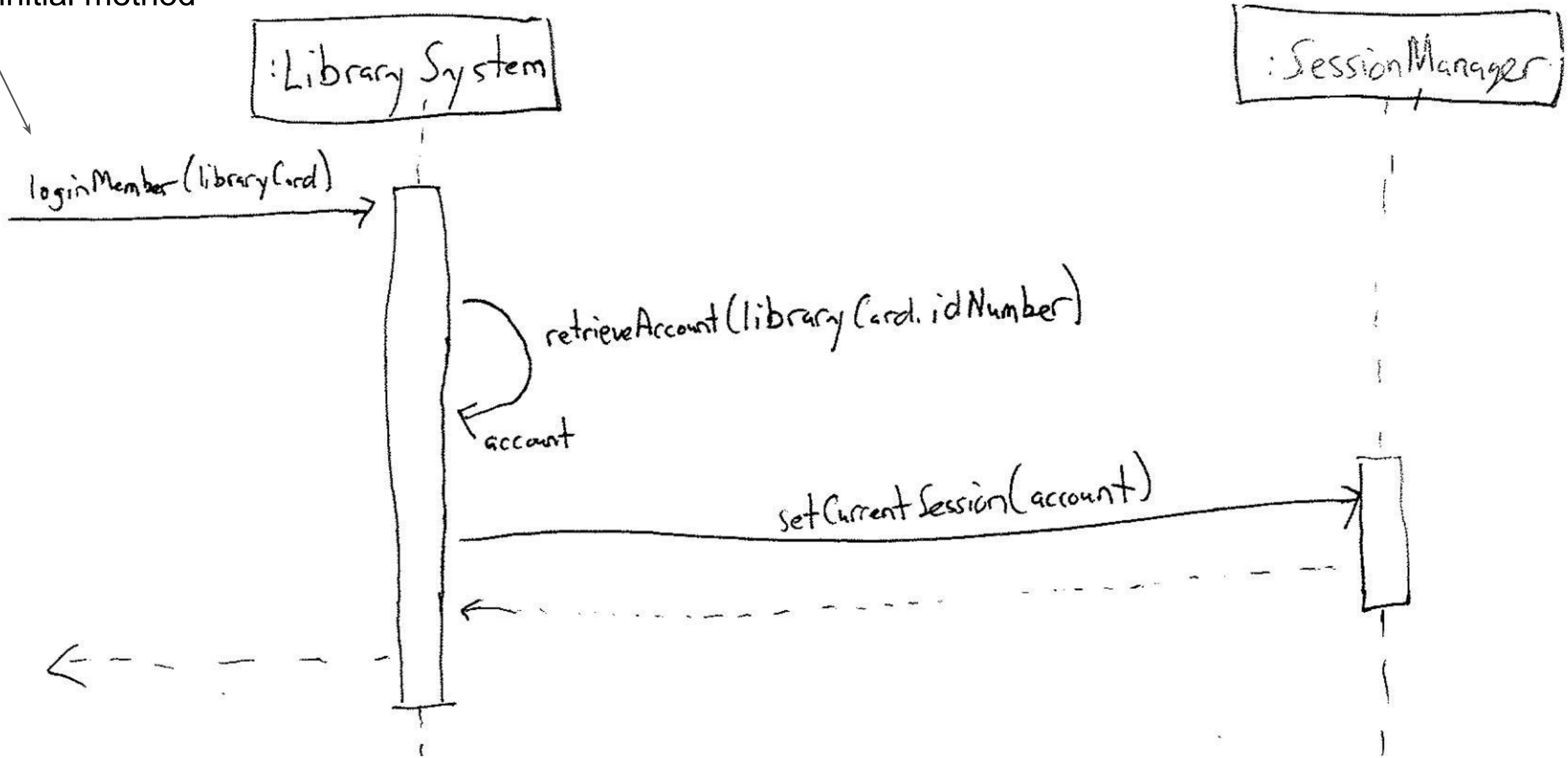
Interactions between objects

Two common notations: sequence diagrams and communication diagrams

Sequence diagrams like system sequence diagrams, but depicting interactions between objects/classes

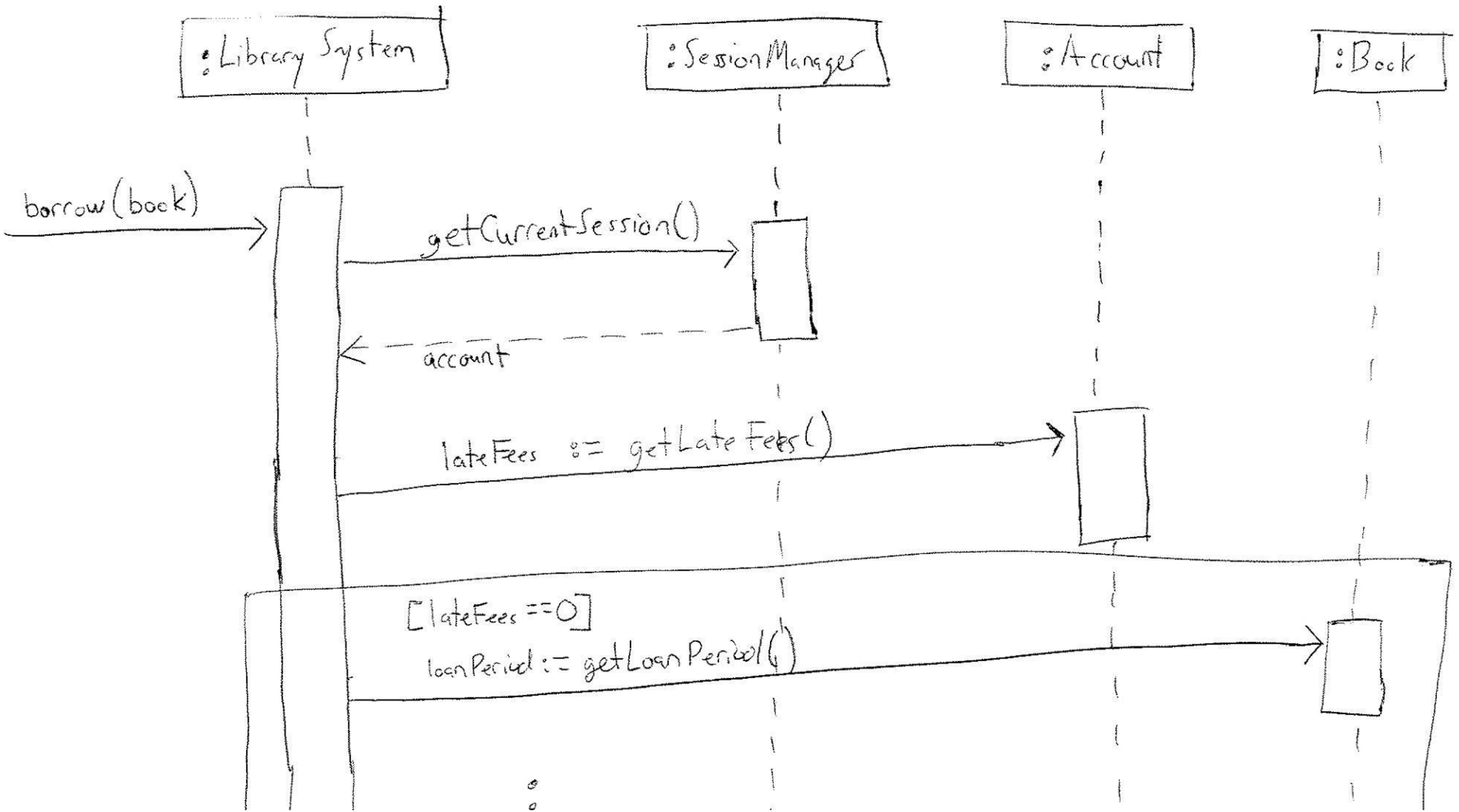


always start with
an initial method



Interaction Diagram Practice:

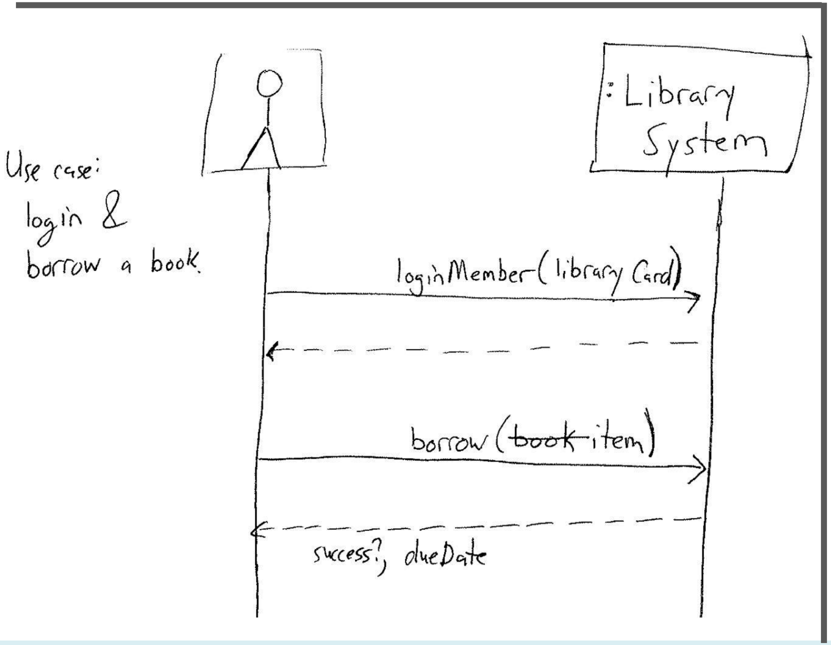
<u>Use case scenario</u>: …and borrow a book. After confirming that the member has no unpaid late fees, the library system should determine the book's due date by adding its loan period to the current day, and record the book and its due date as a borrowed item in the member's library account.

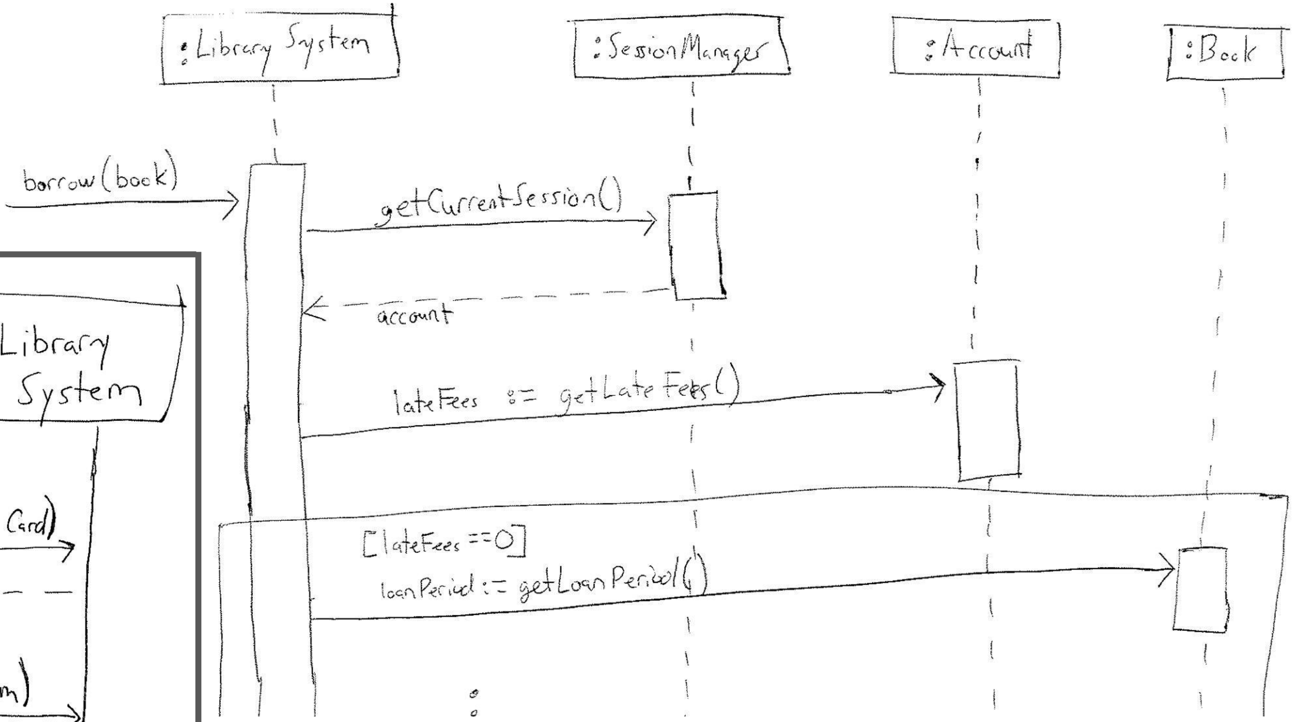

:LibrarySystem
:SessionManager
:Account
:Book
borrow(book)
getCurrentSession()
account
lateFees := getLateFees()
[lateFees == 0]
loanPeriod := getLoanPeriod()

Interaction diagrams help evaluate design alternatives

- Explicitly consider design alternatives
- For each, sketch the interactions implied by the design choice
    - Interactions correspond to the components' APIs

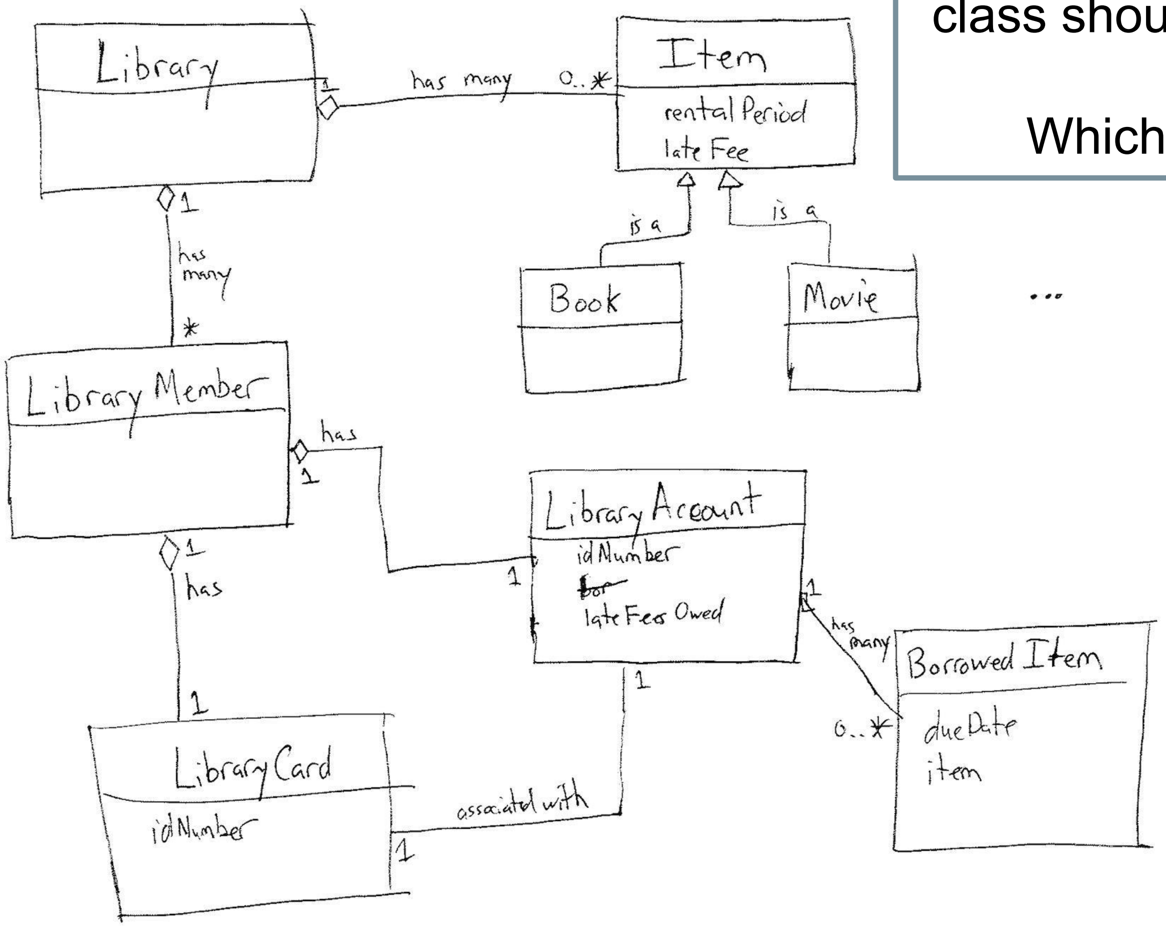
Sys seq diag. (left) vs interaction diag. (right)





Object-Level Design

35



Considering the Library problem, which class should **know** which items have been borrowed by a user?
Which should **compute** late fees?

Doing and Knowing *Responsibilities*

Responsibilities are related to the obligations of an object in terms of its behavior.

Doing responsibilities of an object include:

- Doing something itself, such as creating an object or doing a calculation
- Initiating action in other objects
- Controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- Knowing about private encapsulated data
- Knowing about related objects
- Knowing about things it can derive or calculate

Doing and Knowing *Responsibilities*

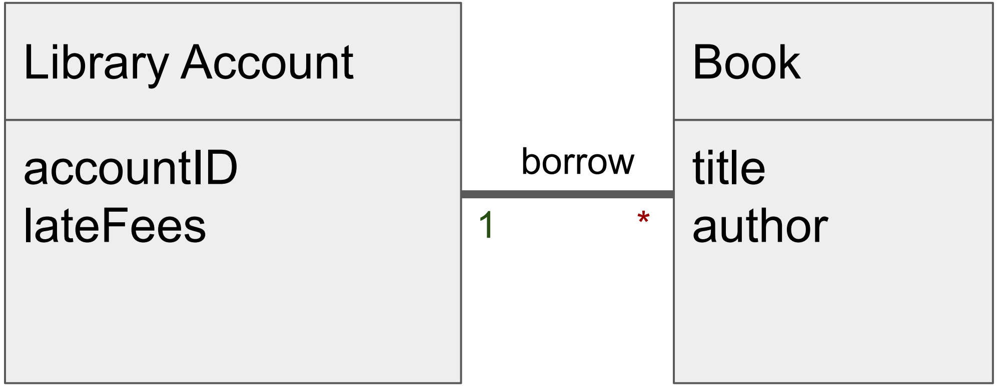
- Object design is not as clear-cut as domain modeling
  - The challenge in domain modeling is being very precise
  - As we get closer to implementation, we need to make choices
    - Where to put data, methods.
    - Never quite 1-to-1 with the real-world concepts.
- Thinking about Assigning Responsibilities helps
  - We'll rely on design principles and heuristics to guide us
  - Including most of GRASP, as set of "General Responsibility Assignment Software Patterns/Principles"

Design Principle:
Low Representational Gap

Low Representational Gap

Identified concepts provide inspiration for classes in the implementation

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

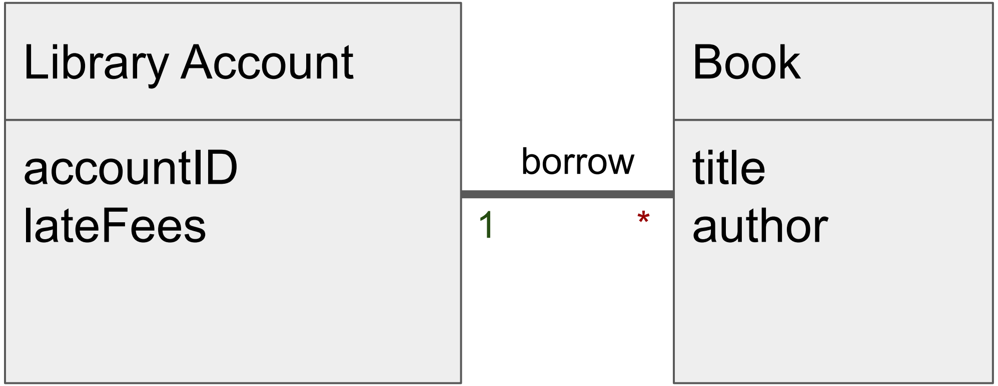



```
class Account {
    id: Int;
    lateFees: Int;
    borrowed: List<Book>;
    boolean borrow(Book) { … }
    void save();
}
class Book { … }
```

Low Representational Gap

Identified concepts provide inspiration for classes in the implementation

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



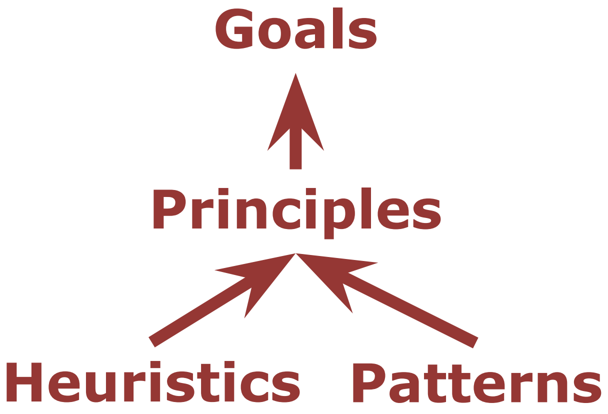

```
class LibraryDatabase {
    Map<Int, List<Int>>
        borrowedBookIds;
    Map<Int, Int> lateFees;
    Map<Int, String>
        bookTitles;
}
class DatabaseRow { … }
```

Designs with Low Representational Gap

- Create software class for each domain class, create corresponding relationships
- Design goal: Design for change
- This is only a starting point!
  - Not all domain classes need software correspondence
  - Pure fabrications might be needed
  - Other principles often more important

Design Goals, Principles, and Patterns

- Design Goals
  - Design for change, understanding, reuse, division of labor, …
- Design Principle
  - Low coupling, high cohesion
  - Low representational gap
- Design Heuristics
  - Law of demeter
  - Information expert
  - Creator
  - Controller



DESIGN PRINCIPLE: LOW COUPLING

44

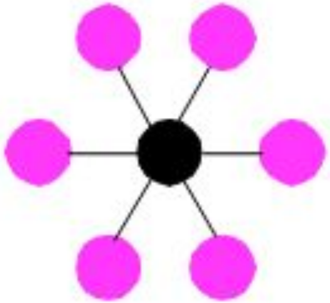
Design Principle: Low Coupling

A module should depend on as few other modules as possible

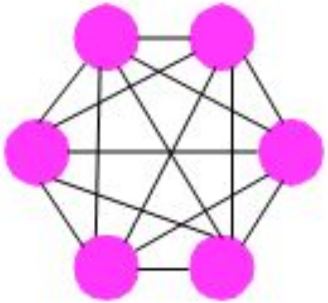
- Enhances understandability (*design for understanding*)
  - Limited understanding of context, easier to understand in isolation
- Reduces the cost of change (*design for change*)
  - Little context necessary to make changes
  - When a module interface changes, few modules are affected (reduced rippling effects)
- Enhances reuse (*design for reuse*)
  - Fewer dependencies, easier to adapt to a new context

Topologies with different coupling

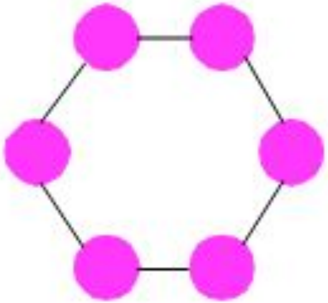
Types of module interconnection structures



(A)



(B)



(C)

High Coupling is undesirable

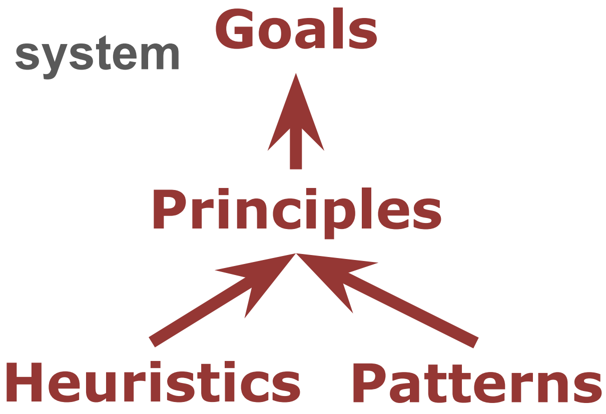
- Element with low coupling depends on few other elements
  - Elements == classes, subsystems, …
  - “few" is context-dependent
- A class with high coupling relies on many other classes
  - Changes in related classes force local changes; changes in local class forces changes in related classes (brittle, 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?

```java
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;
}
```

Design Goals, Principles, and Patterns

- Design Goals
  - Design for change, understanding, reuse, division of labor, …
- Design Principle
  - Low coupling, high cohesion
  - Low representational gap
- **Design Heuristics: promoting principle(s) in the system**
  - Law of demeter
  - Information expert
  - Creator
  - Controller



Design Heuristic: 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 (let i of shipment.getBox().getItems())
    shipmentWeight += i.getWeight() …
```

So don’t do this ^ !!

Coupling: Discussion

- 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*
- (Details next time:) 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*
- Coupling is one principle among many
  - Consider cohesion, low repr. gap, and other principles

Design Goals

- Explain how low coupling supports
  - Design for change
  - Design for understandability
  - Design for division of labor
  - Design for reuse
  - …

Design Goals

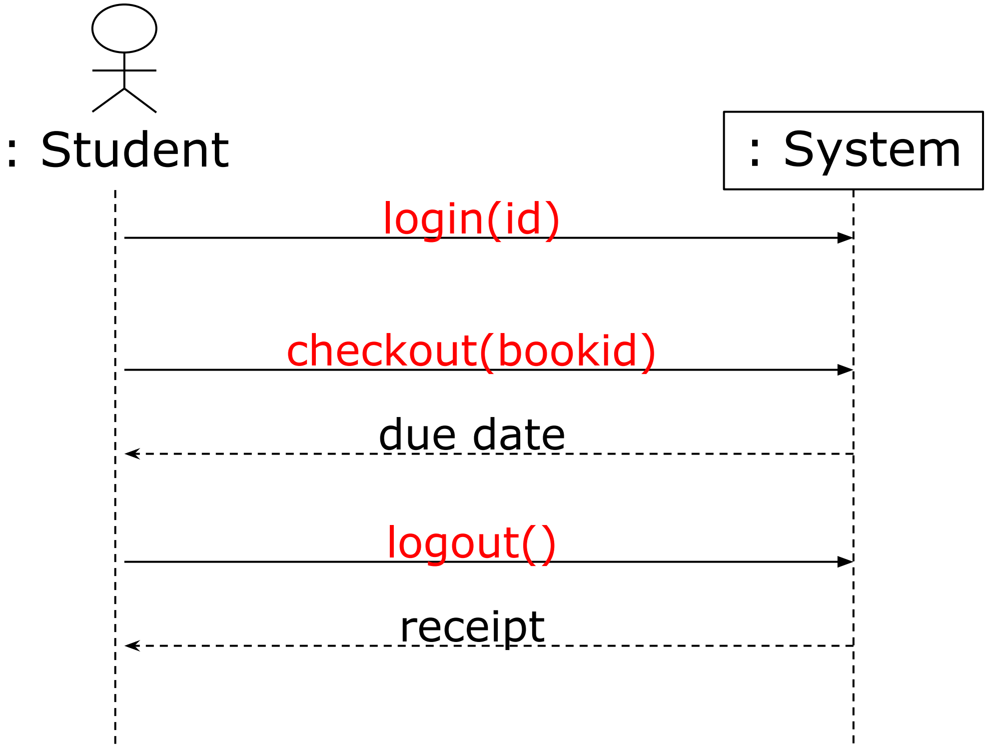
- Design for change
  - Changes easier because fewer dependencies on fewer other objects
  - Changes are less likely to have rippling effects
- Design for understandability
  - Fewer dependencies to understand (e.g., a.getB().getC().foo())
- Design for division of labor
  - Smaller interfaces, easier to divide
- Design for reuse
  - Easier to reuse without complicated dependencies

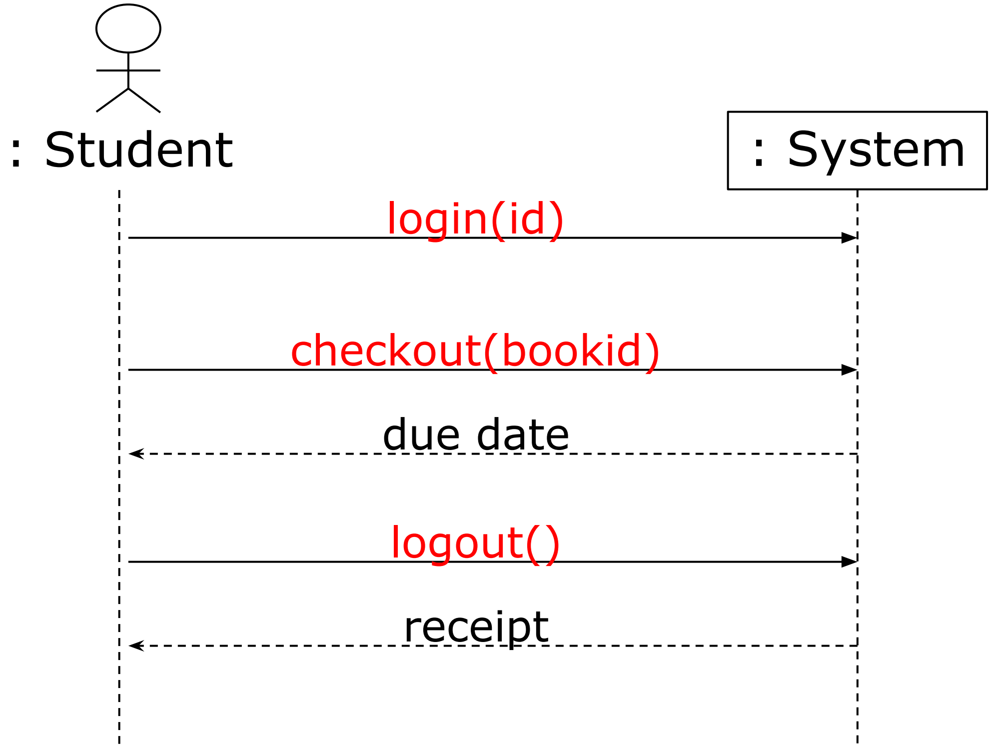
Design Heuristic: CONTROLLER

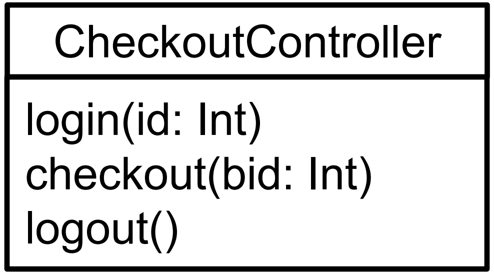
(also DESIGN PATTERN: FAÇADE)

Controller (Design Heuristic)

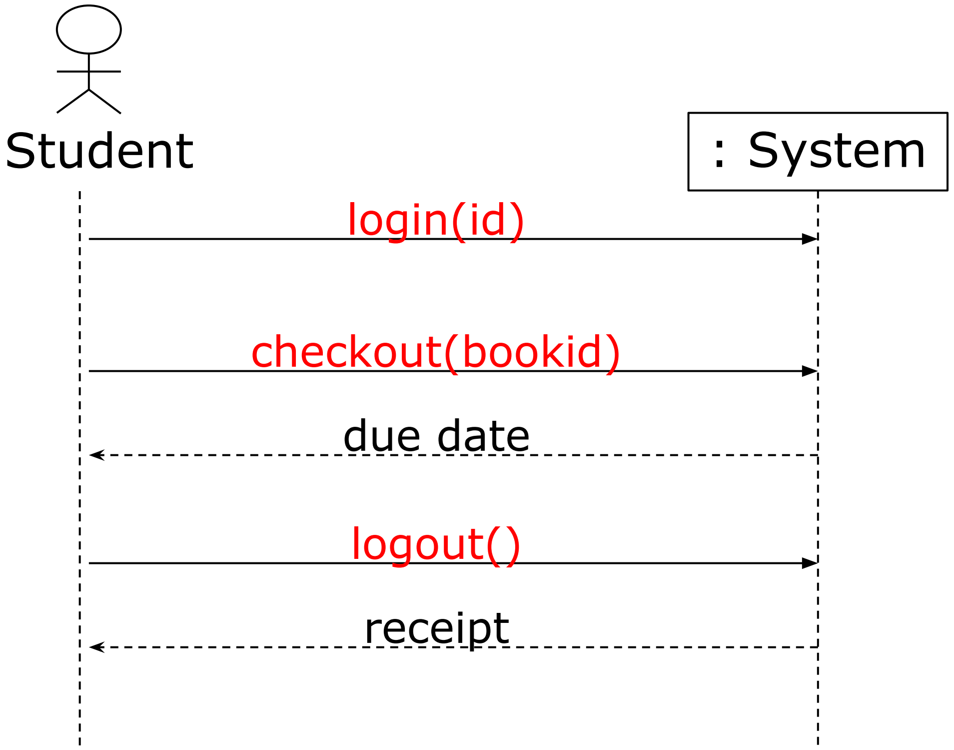
- Problem: What object receives and coordinates a system operation (event)?
- Solution: Assign the responsibility to an object representing
  - The overall system, device, or subsystem (façade controller), or
  - A use case scenario within which the system event occurs (use case controller)
- Process: Derive from system sequence diagram (key principles: Low representational gap and high cohesion)


: Student
: System
login(id)
checkout(bookid)
due date
logout()
receipt

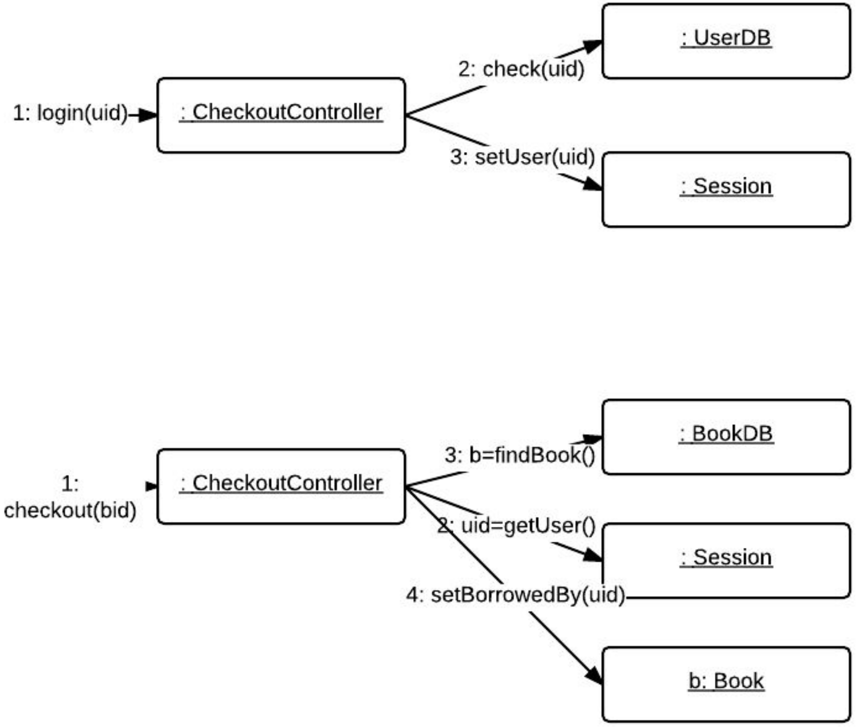

: Student
: System
login(id)
checkout(bookid)
due date
logout()
receipt


CheckoutController
login(id: Int)
checkout(bid: Int)
logout()

Requirements Analysis



Object-Level Design



Controller: Discussion

- A Controller is a coordinator
  - Does not do much work itself
  - Delegates to other objects
- Façade controllers suitable when not "too many" system events
  - -> One overall controller for the system
- Use case controller suitable when façade controller "bloated" with excessive responsibilities (low cohesion, high coupling)
  - -> Several smaller controllers for specific tasks
- Closely related to Façade design pattern (future lecture)

Controller: Design Tradeoffs

Decreases coupling

- User interface and domain logic are decoupled from each other
  - Understandability: can understand these in isolation, leading to:
  - Evolvability: both the UI and domain logic are easier to change
- Both are coupled to the controller, which serves as a mediator, but this coupling is less harmful
  - The controller is a smaller and more stable interface
  - Changes to the domain logic affect the controller, not the UI
  - The UI can be changed without knowing the domain logic design

Supports reuse

- Controller serves as an interface to the domain logic
- Smaller, explicit interfaces support evolvability

But, bloated controllers increase coupling and decrease cohesion; split if applicable

Controller in Flash Cards Project?

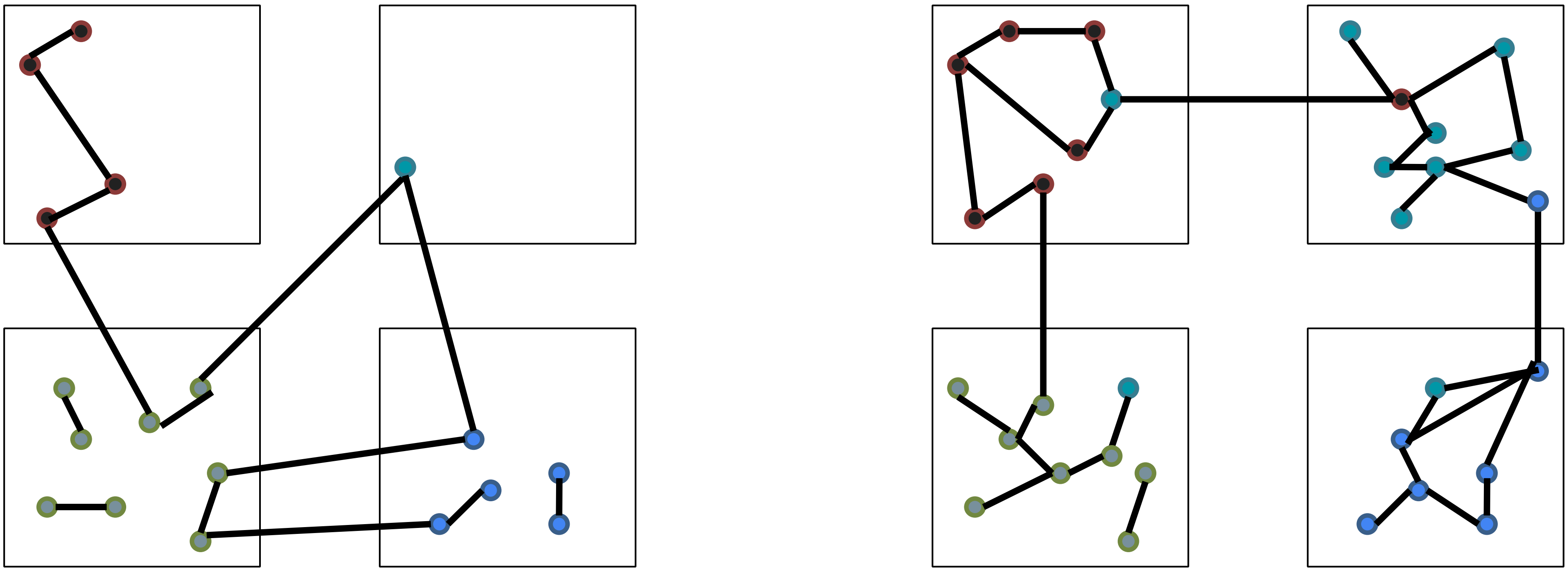
DESIGN PRINCIPLE: HIGH COHESION

(OR SINGLE RESPONSIBILITY PRINCIPLE)

Design Principle: Cohesion

A module should have a small set of related responsibilities

- Enhances understandability (design for understandability)
  - A small set of responsibilities is easier to understand
- Enhances reuse (design for reuse)
  - A cohesive set of responsibilities is more likely to recur in another application



```
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
    }
}
```

Anti-Pattern: God Object

```
class Chat {
    List<String> channels;
    Map<String, List<Msg>> messages;
    Map<String, String> accounts;
    Set<String> bannedUsers;
    File logFile;
    File bannedWords;
    URL serverAddress;
    Map<String, Int> globalSettings;
    Map<String, Int> userSettings;
    Map<String, Graphic> smileys;
    CryptStrategy encryption;
    Widget sendButton, messageList;
}
```

Anti-Pattern: God Object

```
class Chat {
    List<String> channels;
    Map<String, List<Msg>> messages;
    Map<String, String> accounts;
    Set<String> bannedUsers;
    File logFile;
    File bannedWords;
    URL serverAddress;
    Map<String, Int> globalSettings;
    Map<String, Int> userSettings;
    Map<String, Graphic> smileys;
    CryptStrategy encryption;
    Widget sendButton, messageList;
```

```
class Chat {
    Content content;
    AccountMgr accounts;
    File logFile;
    ConnectionMgr conns;
}
class ChatUI {
    Chat chat;
    Widget sendButton, …;
}
class AccountMgr {
    … acounts, bannedUsr…
}
```

Cohesion in Graph Implementations

```
class Graph {
    Node[] nodes;
    boolean[] isVisited;
}
class Algorithm {
    int shortestPath(Graph g, Node n, Node m) {
        for (int i; …)
            if (!g.isVisited[i]) {
                …
                g.isVisited[i] = true;
            }
        }
        return v;
    }
}
```

Is this a good implementation?

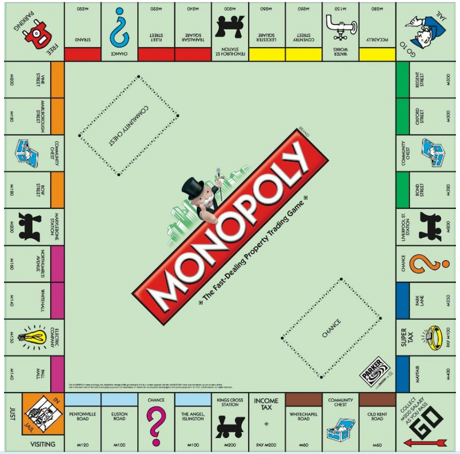
Cohesion in Graph Implementations

```
class Graph {
    Node[] nodes;
    boolean[] isVisited;
}
class Algorithm {
    int shortestPath(Graph g, Node n, Node m) {
        for (int i; …)
            if (!g.isVisited[i]) {
                …
                g.isVisited[i] = true;
            }
        }
        return v;
    }
}
```

Graph is tasked with not just data, but also algorithmic responsibilities

Monopoly Example

Which design has higher cohesion?




```
class Player {
    Board board;
    /* in code somewhere… */ this.getSquare(n);
    Square getSquare(String name) { // named monopoly squares
        for (Square s: board.getSquares())
            if (s.getName().equals(name))
                return s;
        return null;
}}
```

```
class Player {
    Board board;
    /* in code somewhere… */ board.getSquare(n);
}
class Board{
    List<Square> squares;
    Square getSquare(String name) {
        for (Square s: squares)
            if (s.getName().equals(name))
                return s;
        return null;
}}
```

Hints for Identifying Cohesion

- Use one color per concept
- Highlight all code of that concept with the color
- => Classes/methods
  should have few colors

Hints for Identifying Cohesion

- There is no clear definition of what a “concept” is
- Concepts can be split into smaller concepts
  - Graph with search vs. Basic Graph + Search Algorithm vs. Basic Graph + Search Framework + Concrete Search Algorithm etc
- Requires engineering judgment

Cohesion: Discussion

Very Low Cohesion: A Class is solely responsible for many things in very different functional areas

Low Cohesion: A class has sole responsibility for a complex task in one functional area

High Cohesion: A class has moderate responsibilities in one functional area and collaborates with classes to fulfil tasks

Advantages of high cohesion

- Classes are easier to maintain
- Easier to understand
- Often support low coupling
- Supports reuse because of fine grained responsibility

Rule of thumb: a class with high cohesion has relatively few methods of highly related functionality; does not do too much work

Coupling vs Cohesion (Extreme cases)

All code in one class/method

- very low coupling, but very low cohesion

Every statement separated

- very high cohesion, but very high coupling

Find good tradeoff; consider also other principles, e.g., low representational gap

Cohesion in Flash Cards Project?

Design Heuristic: INFORMATION EXPERT

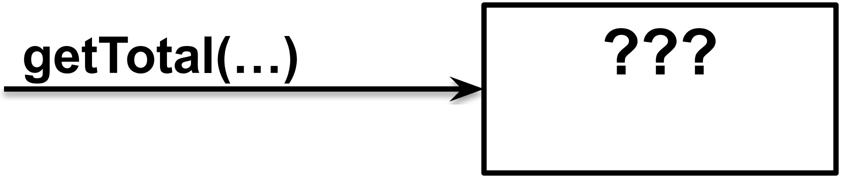
Information Expert (Design Heuristic)

- Heuristic: **Assign a responsibility to the class that has the information necessary to fulfill the responsibility**
- Typically follows common intuition
- Software classes instead of Domain Model classes
    - If software classes do not yet exist, look in Domain Model for fitting abstractions (-> correspondence)
- Design process: Derive from domain model (key principles: Low representational gap and low coupling)

```
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;
}
```

Which class has all the information to compute the shipment's weight?

Who should be responsible for knowing the grand total of a sale?




TARGET

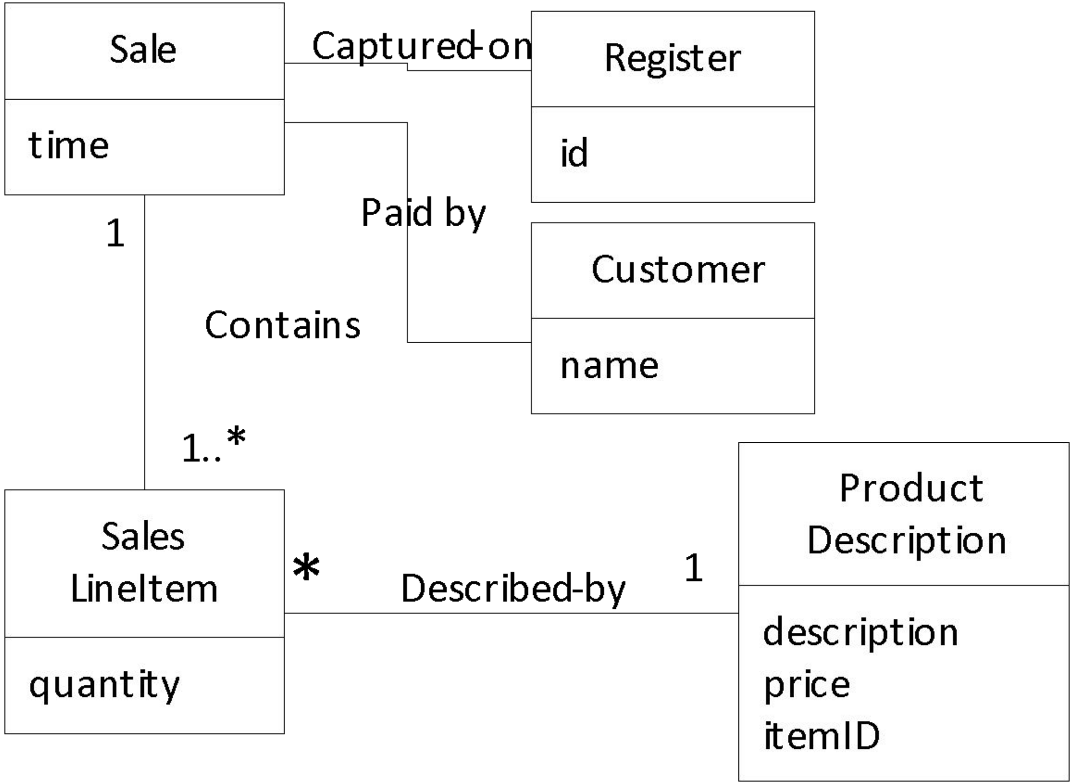
07/20/02 10:40 AM
RETURN BEFORE 10/18/02

GIVING A GIFT? Include a gift receipt!
A receipt dated within 90 days is
required for all returns & exchanges.

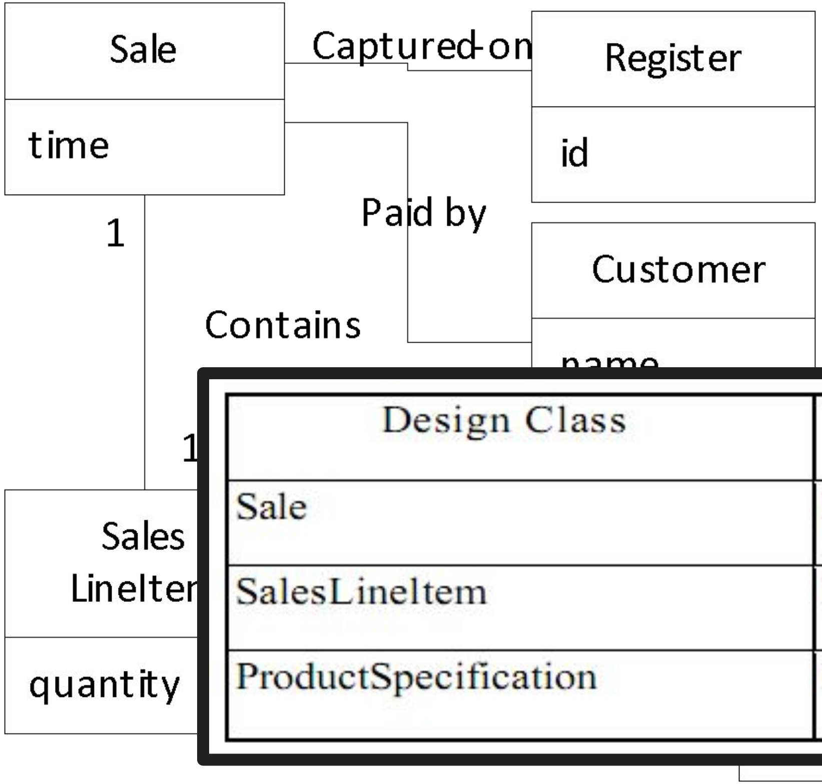
001	212460341	GENERAL MILL	FN	2.89
002	071100015	RITZ CRCKERS	FN	2.54
003	212480045	PILLSBURY	FN	1.29
004	212140336	BC CHKN HLPR	FN	1.69
005	071100009	WHEAT THINS	FN	1.99
006	203700129	OCEAN SPRAY	FN	1.89
007	212080143	W BONE DRSNG	FN	1.79
008	284010136	DIGIORNO	FN	2.54
009	003060057	DAWN	T	1.99
010	071090122	CHIPS AHOY	FN	2.54
		SUBTOTAL		21.15
		T= 5.000% TAX		.10
		TOTAL		21.25

Register
Sale
LineItem
Product Descr.

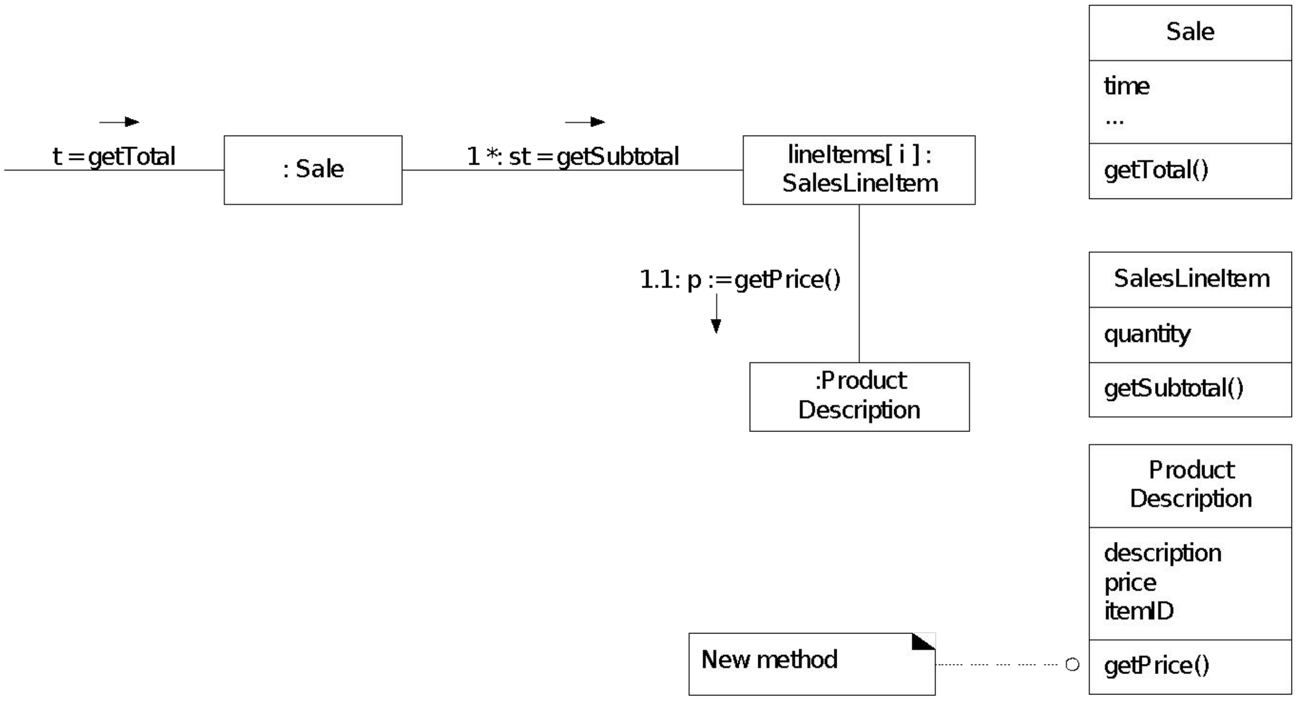
Who should be responsible for knowing the grand total of a sale?



Who should be responsible for knowing the grand total of a sale?



Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price



t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i] :
SalesLineItem
1.1: p := getPrice()
:Product
Description
Sale
time
...
getTotal()
SalesLineItem
quantity
getSubtotal()
Product
Description
description
price
itemID
getPrice()
New method

Information Expert → "Do It Myself Strategy"

- Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents
  - A sale does not tell you its total; it is an inanimate thing
- In OO design, all software objects are "alive" or "animated," and they can take on responsibilities and do things.
- They do things related to the information they know.

Information Experts in FlashCards™©®?

Who knows the text on a card?

Who checks correctness of an answer?

Who processes command-line options?

Who stores past answers?

Who knows how to flip cards?

Who tracks which achievements have been achieved?

Design Heuristic: CREATOR

Creator (Design Heuristic)

Problem: Who creates an A?

Solution: Assign class responsibility of creating instance of class A to B if

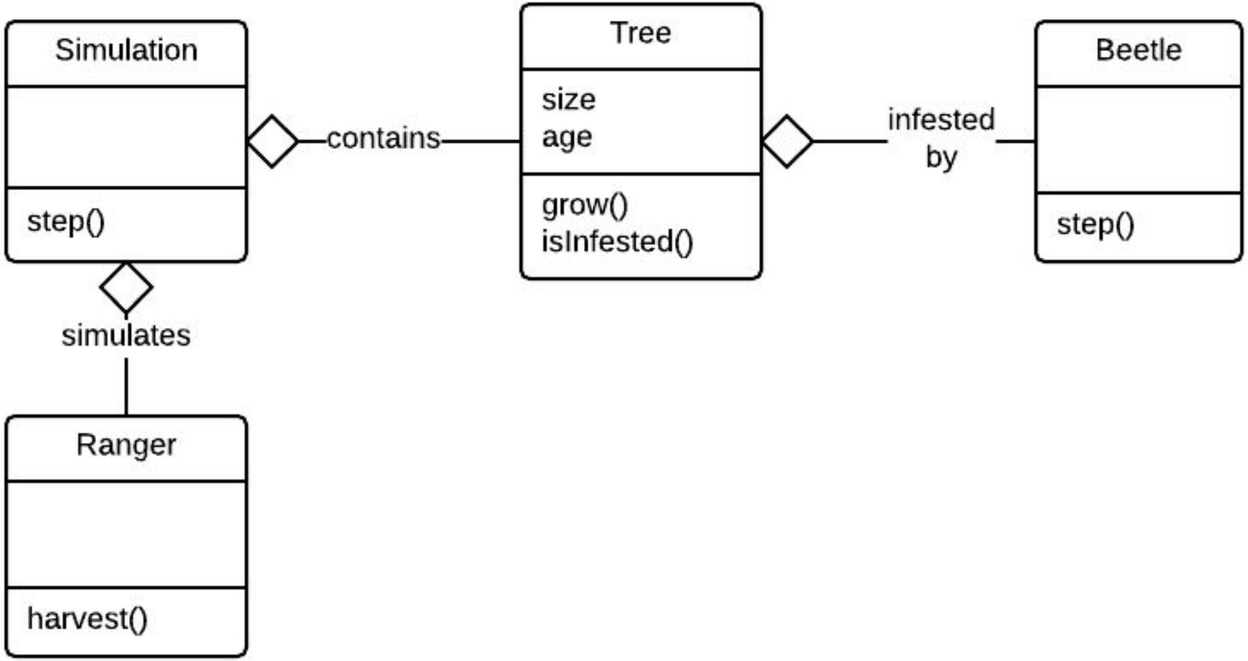
- B aggregates A objects, B contains A objects, B records instances of A objects, B closely uses A objects, B has the initializing data for creating A objects (the more the better)
- Where there is a choice, prefer B aggregates or contains A objects

Key idea: Creator needs to keep reference anyway and will frequently use the created object

Process: Extract from domain model, interaction diagrams (key principles: Low coupling and low representational gap)

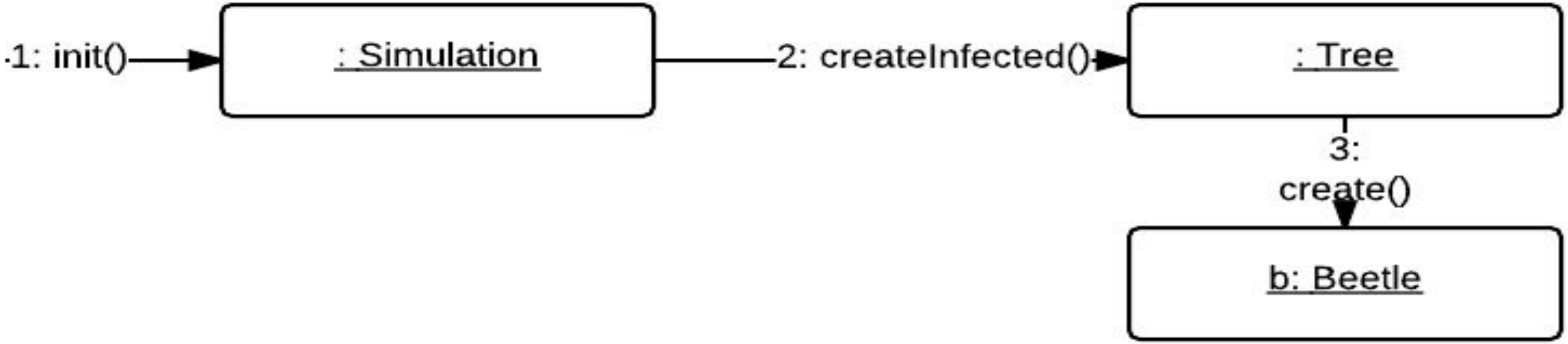
Creator: Example

- Who is responsible for creating Beetle objects?



Creator: Example

- Who is responsible for creating Beetle objects?
  - Creator pattern suggests Tree
- Interaction diagram:



Creator (GRASP)

- Problem: Assigning responsibilities for creating objects
  - Who creates Nodes in a Graph?
  - Who creates instances of SalesItem?
  - Who creates Children in a simulation?
  - Who creates Tiles in a Monopoly game?
    - AI? Player? Main class? Board? Meeple (Dog)?

Creator: Discussion of Design Goals/Principles

Promotes **low coupling**, **high cohesion**

- Class responsible for creating objects it needs to reference
- Creating the objects themselves avoids depending on another class to create the object

Promotes **evolvability** (design for change)

- Object creation is hidden, can be replaced locally

Contra: sometimes objects must be created in special ways

- Complex initialization
- Instantiate different classes in different circumstances
- *Then* ***cohesion*** *suggests putting creation in a different object*: see *design patterns* such as builder, factory method

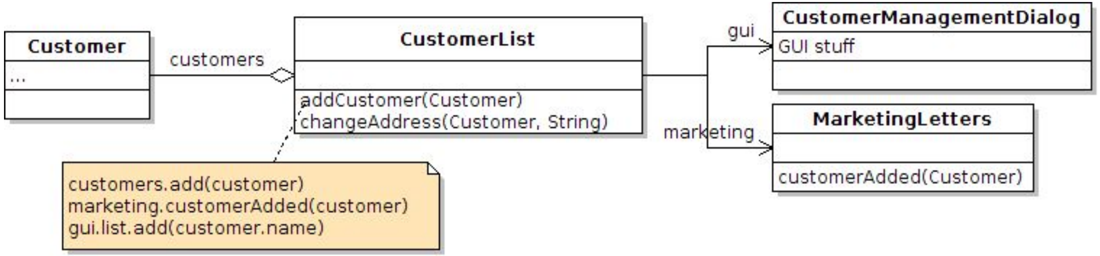
Creator in Flash Cards Project

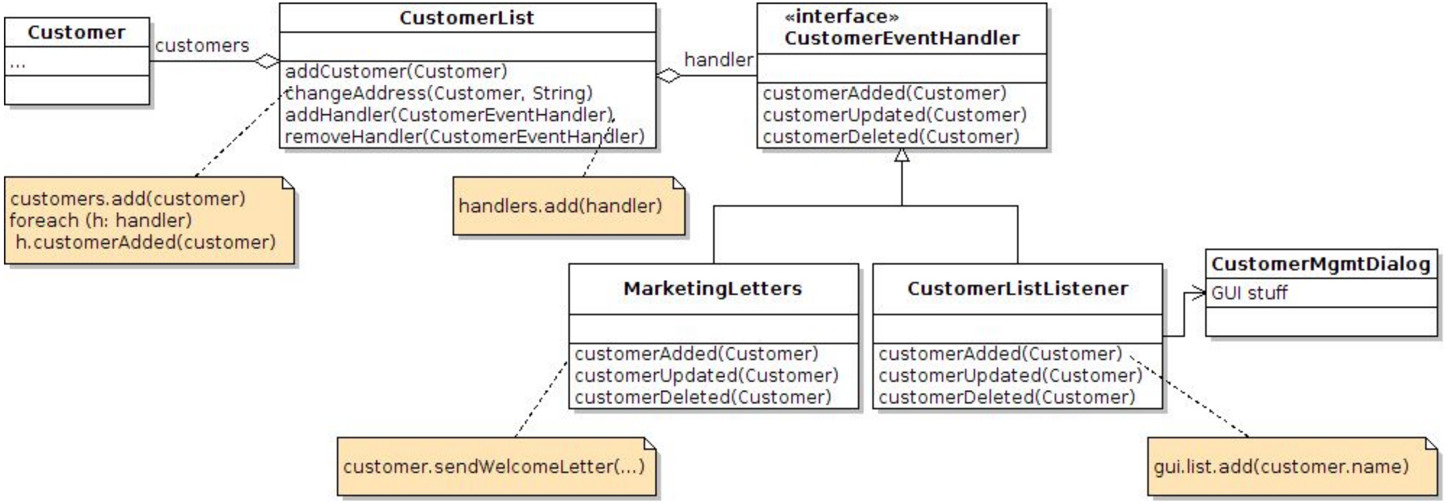
Who creates cards?

Who creates a card deck?

Who creates achievements?

Which design is better? Argue with design goals, principles, heuristics, and patterns that you know





* old midterm question

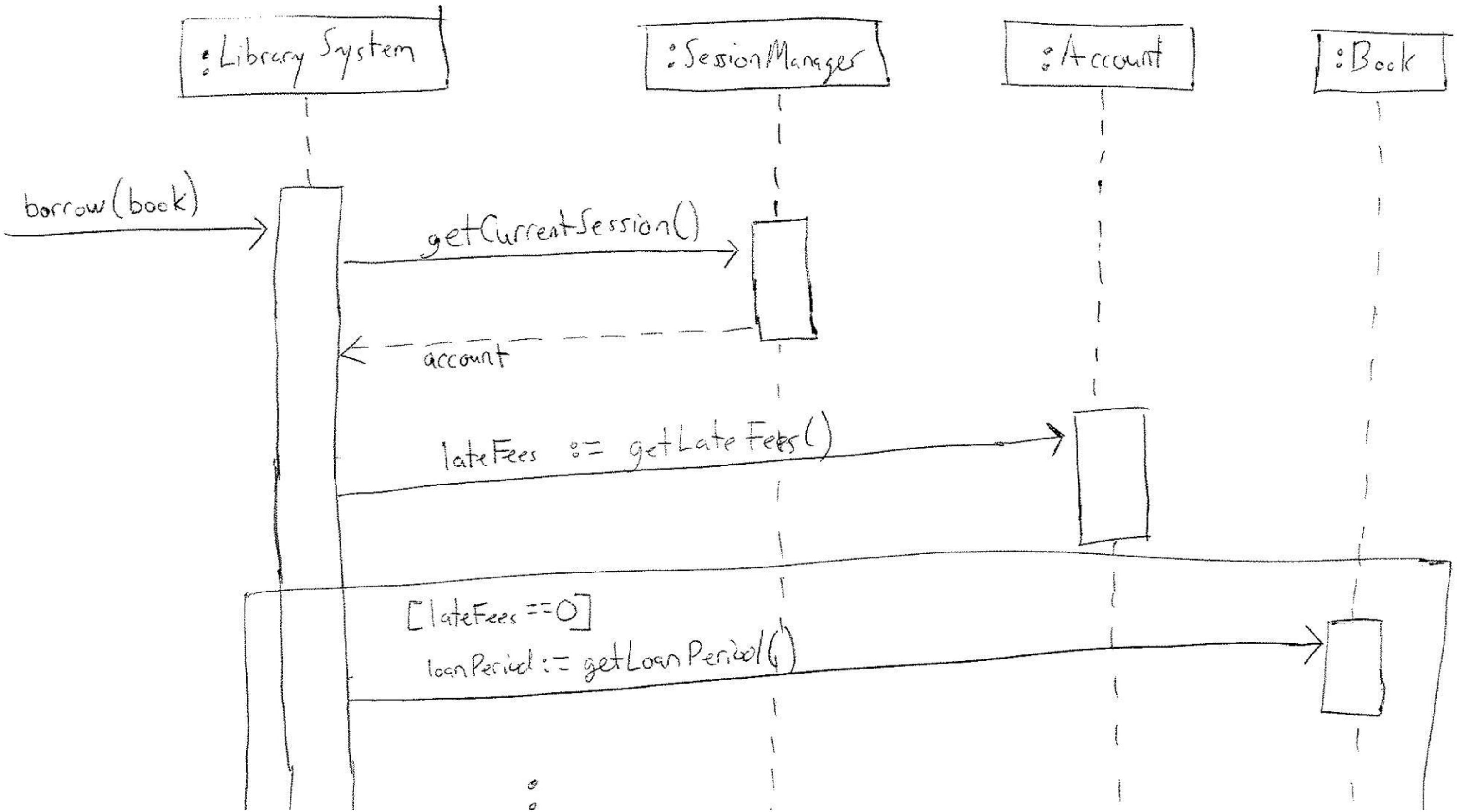
Other Design Heuristics

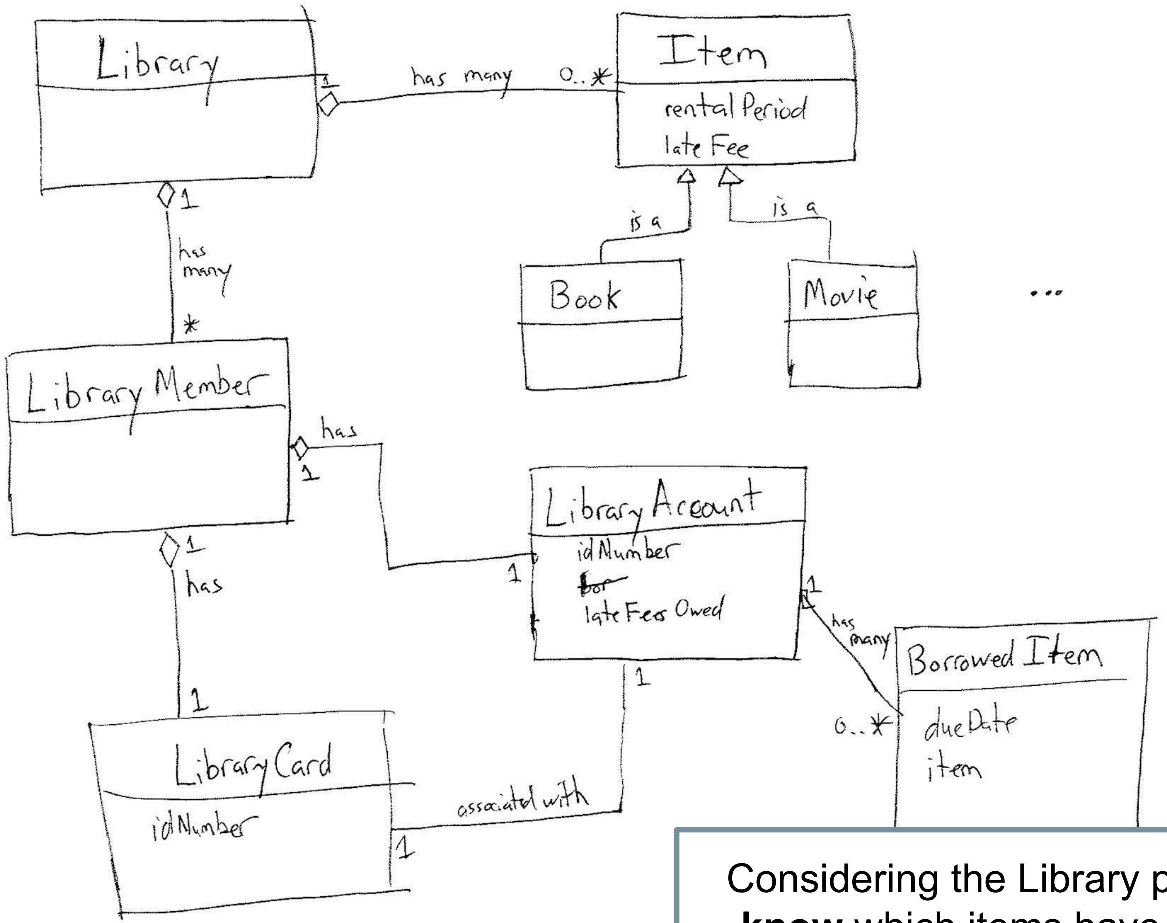
In future lectures:

- Minimize mutability
- Minimize conceptual weight
- Favor composition/delegation over inheritance
- Use indirection to reduce coupling
- …

Object-level artifacts of this design process

- Object interaction diagrams add methods to objects
  - Can infer additional data responsibilities
  - Can infer additional data types and architectural patterns
- Object model aggregates important design decisions
  - Is an implementation guide


:Library System
:SessionManager
:Account
:Book
borrow(book)
getCurrentSession()
account
lateFees := getLateFees()
[lateFees == 0]
loanPeriod := getLoanPeriod()



Considering the Library problem, which class should **know** which items have been borrowed by a user? Which should **compute** late fees?

Take-Home Messages

Design is driven by quality attributes

- Evolvability, separate development, reuse, performance, …

Design principles provide guidance on achieving qualities

- Low coupling, high cohesion, high correspondence, …

GRASP design heuristics promote these principles

- Creator, Expert, Controller, …